2

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

AUTOMATED DESIGN OF A MICROPROGRAMMED CONTROLLER FOR A FINITE STATE MACHINE

by

James Edward Harmon

June 1988

Thesis Co-advisor:	D. E. Kirk
Thesis Co-advisor:	H. H. Loomis, Jr.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Naval Postgraduate School — 6b OFFICE SYMBOL (If applicable) 62	7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000	7b. ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000
8a NAME OF FUNDING/SPONSORING ORGANIZATION — 8b OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State and ZIP Code)	10 SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO.

11 TITLE (Include Security Classification)
AUTOMATED DESIGN OF A MICROPROGRAMMED CONTROLLER FOR A FINITE STATE MACHINE

12 PERSONAL AUTHOR(S)
HARMON, James E.

13a. TYPE OF REPORT	13b TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Master's Thesis	FROM ____ TO ____	1988 June	231

16 SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	microprogrammed controller, VLSI design, silicon compiler, finite state machine

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

A Scalable Complementary Metal Oxide Semiconductor (SCMOS) microprogrammed controller for the Monterey Silicon Compiler (MSC) is implemented in the LISP programming language. The internal organization of MSC and the architecture of Very Large Scale Integrated (VLSI) circuits generated by MSC are discussed.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a NAME OF RESPONSIBLE INDIVIDUAL H. H. Loomis, Jr.	22b TELEPHONE (Include Area Code) (408)646-3214 — 22c OFFICE SYMBOL 62Lm

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Automated Design of a Microprogrammed Controller for a Finite State Machine

by

James Edward Harmon
Lieutenant Commander, United States Navy
B.S., University of New Mexico, 1975

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
June 1988

Author: [illegible]
J. E. Harmon

Approved by: [illegible]
D. E. Kirk, Thesis Co-Advisor

[illegible]
H. H. Loomis, Thesis Co-Advisor

[illegible]
J. P. Powers, Chairman, Department of Electrical and Computer Engineering

[illegible]
Gordon E. Schacher,
Dean of Science and Engineering

ABSTRACT

A Scalable Complementary Metal Oxide Semiconductor (SCMOS) microprogrammed controller for the Monterey Silicon Compiler (MSC) is implemented in the LISP programming language. The internal organization of MSC and the architecture of Very Large Scale Integrated (VLSI) circuits generated by MSC are discussed. Keywords: theses (KR)

TABLE OF CONTENTS

I. INTRODUCTION ---------- 1

A. SILICON COMPILERS ---------- 1

B. MONTEREY SILICON COMPILER (MSC) PROJECT ---------- 3

C. MICROPROGRAMMED CONTROLLER ---------- 4

D. NAME CONVENTIONS ---------- 5

E. CONTENT DESCRIPTION ---------- 6

1. Analysis Phase ---------- 6

2. Design Phase ---------- 6

3. Implementation Phase ---------- 7

4. Test Phase ---------- 7

5. Maintenance Phase ---------- 7

II. MSC TARGET ARCHITECTURE ---------- 8

A. PHYSICAL LAYOUT ---------- 8

1. Data Path ---------- 10

2. Control Logic Array ---------- 16

B. FINITE STATE MACHINE (FSM) ARCHITECTURE ---------- 17

1. Moore FSM ---------- 18

2. Mealy FSM ---------- 19

3. Moore FSM Timing ---------- 19

4. Mealy FSM Timing ---------- 21

5. State Transition Diagram ---------- 21

6. Moore FSM State Transition Diagram ---------- 22

7. Mealy FSM State Transition Diagram ---------- 24

C. MSC FSM STRUCTURE ---------------------------- 25

D. MULTIPLE FSM INTERACTION ---------------------- 27

1. nMOS Multiplexer Wired AND Logic ---------- 28

2. SCMOS Multiplexer Logic ------------------ 31

E. ARCHITECTURAL REQUIREMENTS -------------------- 32

III. MSC INTERNAL STRUCTURE ---------------------------- 33

A. DATA PRIMITIVES ------------------------------- 33

B. DATA STRUCTURES ------------------------------- 34

C. CONTROL AND DATA FLOW ------------------------- 36

D. PROGRAM COMPONENTS ---------------------------- 38

1. Program lincoln.l ------------------------- 38

2. Program c-routines.c ---------------------- 39

3. Program L5.l ------------------------------ 40

4. Program defstructs.l ---------------------- 40

5. Program front-page.l ---------------------- 40

6. Program prepass.l ------------------------- 41

7. Program library --------------------------- 41

8. Program organelles.l ---------------------- 43

9. Program general.l ------------------------- 43

10. Program extract.l ------------------------- 44

a. Get-sequencers-from-component-list ---- 45

b. Get-sequencers-required-definitions --- 47

c. Extract-component-list ---------------- 50

d. Post-process -------------------------- 50

11. Program order.l --------------------------- 51

12. Program frame.l --------------------------- 52

13. Program data-path.l ---------------------- 53
14. Program control.l ------------------------- 55
15. Program flags.l --------------------------- 55
16. Program pads.l ---------------------------- 55
a. Program padgen.l ---------------------- 56
b. Pad File pad20b ----------------------- 56
c. Pad File rinout ----------------------- 56
17. Program interpret.l ---------------------- 56
18. Makefile ---------------------------------- 57
E. MSC INTERNAL STRUCTURE REQUIREMENTS ----------- 58
IV. CONTROLLER GOALS AND REQUIREMENTS ----------------- 59
A. MSC CONTROLLER GOALS -------------------------- 59
B. MSC CONTROLLER REQUIREMENTS ------------------- 60
V. ARCHITECTURE TRADE-OFF STUDIES -------------------- 62
A. LOGIC STRUCTURE TRADE-OFF STUDIES ------------- 63
1. Structural Requirements Analysis ---------- 63
a. Static CMOS Complementary Logic ------- 64
b. Pseudo-nMOS Logic --------------------- 64
c. Dynamic CMOS Logic -------------------- 66
d. Clocked CMOS Logic -------------------- 68
e. CMOS Domino Logic --------------------- 69
f. Cascade Voltage Switch Logic ---------- 71
g. Modified Domino Logic ----------------- 72
h. Transmission Gate Logic --------------- 73
i. Acceptable Structures ----------------- 77
2. Structural Goals Analysis ----------------- 77

a. Static CMOS Complementary Logic ------- 77

b. Pseudo-nMOS Logic --------------------- 78

c. Static Cascade Voltage Switch Logic --- 78

d. Complementary Transmission Gate Logic - 79

3. Structure Selection ------------------------ 80

B. LOGIC ORGANIZATION TRADE-OFF STUDIES ---------- 80

1. Data Path Units ---------------------------- 81

a. Sequencer Types ------------------------ 81

b. Data Path Unit Advantages. ------------ 86

c. Data Path Unit Alternatives ----------- 87

2. Standard Cell or Microprogram ROM --------- 87

a. Standard Cell ------------------------- 87

b. Microprogram ROM ---------------------- 88

C. LOGIC STRUCTURE AND ORGANIZATION SELECTION ---- 89

VI. CONTROLLER DESIGN ---------------------------------- 92

A. FUNCTIONAL REQUIREMENTS PARTITIONS ------------ 92

B. UNIT IDENTIFICATION --------------------------- 93

1. Data Path Units --------------------------- 93

2. Microprogram ROM Units -------------------- 94

a. ROM Array ----------------------------- 94

b. Row Decoder --------------------------- 94

c. Conditional --------------------------- 95

d. Buffer -------------------------------- 96

C. FLOOR PLAN ----------------------------------- 96

1. Controller Floor Plan --------------------- 97

2. Microprogram ROM Floor Plan -------------- 99

D. CELL LOGIC FUNCTIONS ------------------------- 100
1. ROM Array Cells -------------------------- 100
a. ROM1 Cell ---------------------------- 102
b. ROM0 Cell ---------------------------- 103
c. ROMnull Cell ------------------------- 103
2. Row Decoder Cells ------------------------ 103
a. DecodeLSB Cell ----------------------- 104
b. DecodeEven0 and DecodeEven1 Cells ---- 104
c. DecodeOdd0 and DecodeOdd1 Cells ------ 108
d. EvenInverter and OddInverter Cells --- 108
e. EvenConnect and OddConnect Cells ----- 111
f. Two-State Cell ----------------------- 111
3. Conditional Cells ------------------------ 114
a. COND Cell ---------------------------- 115
b. CONDtrue Cell ------------------------ 115
c. CONDfeed and TopCONDfeed Cells ------- 116
d. CONDtap Cell ------------------------- 117
e. CONDpass Cell ------------------------ 118
f. CONDnopass Cells --------------------- 118
4. Buffer Cells ----------------------------- 118
a. RowBuffer Cell ----------------------- 118
b. CONDbuffer Cell ---------------------- 120
c. LSBbuffer Cell ----------------------- 121
d. EvenBuffer Cell ---------------------- 122
e. OddBuffer Cell ----------------------- 122
E. CELL INTERNAL STRUCTURE ---------------------- 123

1. ROM Array Cells ---------- 123

a. ROM1 Cell ---------- 123

b. ROM0 Cell ---------- 123

c. ROMnull Cell ---------- 125

2. Row Decoder Cells ---------- 125

a. DecodeLSB, DecodeEven0, and DecodeEven1 Cells ---------- 125

b. DecodeOdd0 and DecodeOdd1 Cells ---------- 127

c. EvenInverter and OddInverter Cells ---------- 130

d. EvenConnect and OddConnect Cells ---------- 131

e. Two-State Cell ---------- 131

3. Conditional Cells ---------- 132

a. COND Cell ---------- 132

b. CONDtrue Cell ---------- 134

c. CONDfeed, TopCONDfeed, CONDtap, CONDpass, and CONDnopass Cells ---------- 134

4. Buffer Cells ---------- 134

a. RowBuffer Cell ---------- 135

b. CONDbuffer Cell ---------- 135

c. LSBbuffer Cell ---------- 136

d. EvenBuffer and OddBuffer Cells ---------- 137

F. DESIGN VERIFICATION ---------- 138

VII. CONTROLLER IMPLEMENTATION ---------- 139

A. MSC SOFTWARE INSTALLATION ---------- 139

B. MSC MAGIC AND SCMOS UPGRADE ---------- 140

C. CELL LAYOUT AND TEST ---------- 141

D. ADDITION OF CELLS TO MSC ---------- 142

E. MSC CONTROLLER EXTENSION ---------------------- 144

1. Object File Modifications ---------------- 145

2. Extract.1 Modifications ------------------ 145

3. Extraction Example ----------------------- 145

4. Control.1 Modifications ------------------ 158

5. Single Bit Data-Path --------------------- 158

F. IMPLEMENTATION STATUS ------------------------ 159

VIII. CONTROLLER TESTING ------------------------------- 161

IX. CONCLUSIONS -------------------------------------- 163

A. CONTROLLER DESIGN ---------------------------- 163

B. MSC STRUCTURE -------------------------------- 163

C. ADDITIONAL RESEARCH -------------------------- 164

APPENDIX A ISI INSTALLATION LOG ------------------------ 165

APPENDIX B SOFTWARE CHANGE PROPOSAL SCP-2 -------------- 169

LIST OF REFERENCES ------------------------------------- 202

BIBLIOGRAPHY --- 204

INDEX -- 205

INITIAL DISTRIBUTION LIST ------------------------------ 213

LIST OF TABLES

TABLE 1.1 FILE NAME EXTENSIONS AND FILE TYPES ------------ 5

TABLE 3.1 DATA PRIMITIVES -------------------------------- 34

TABLE 3.2 SEQUENCER TYPES -------------------------------- 45

TABLE 3.3 COUNTER-STACK SEQUENCER DEFINITIONS ----------- 48

TABLE 3.4 COUNTER-NO-STACK SEQUENCER DEFINITIONS -------- 49

TABLE 3.5 NO-COUNTER-NO-STACK SEQUENCER DEFINITIONS ----- 49

LIST OF FIGURES

Figure 1.1 Silicon Compiler 2

Figure 1.2 Microprogrammed Controller 4

Figure 2.1 MSC Floor Plan 9

Figure 2.2 Data Path Organization 10

Figure 2.3 Register Unit 12

Figure 2.4 Output Port and Internal Port Units 13

Figure 2.5 Bit Unit 14

Figure 2.6 Organelle Unit 15

Figure 2.7 Unit Multiplexer Options 16

Figure 2.8 Moore Finite State Machine 18

Figure 2.9 Mealy Finite State Machine 19

Figure 2.10 Moore FSM Timing 20

Figure 2.11 Mealy FSM Timing 22

Figure 2.12 Moore FSM State Transition Diagram 23

Figure 2.13 Mealy FSM State Transition Diagram 24

Figure 2.14 MSC Finite State Machine 26

Figure 2.15 Multiple FSM Interaction 28

Figure 2.16 Multiplexer Wired AND Logic 30

Figure 3.1 MSC Control Flow 37

Figure 5.1 Static CMOS Complementary Logic 65

Figure 5.2 Pseudo-nMOS Logic 66

Figure 5.3 Dynamic CMOS Logic 67

Figure 5.4 Clocked CMOS Logic 69

Figure 5.5 CMOS Domino Logic 70

Figure 5.6 Cascade Voltage Switch Logic ------------------ 72
Figure 5.7 Transmission Gate Logic ---------------------- 74
Figure 5.8 Transmission Gate Discrete Components -------- 75
Figure 5.9 Transmission Gate Logic Reduced Circuit ------ 76
Figure 5.10 No-Counter-No-Stack Sequencer ---------------- 82
Figure 5.11 Counter-No-Stack Sequencer ------------------- 84
Figure 5.12 Counter-Stack Sequencer ---------------------- 85
Figure 6.1 Microprogram ROM Units ----------------------- 95
Figure 6.2 Controller Floor Plan ------------------------ 97
Figure 6.3 Alternate Controller Floor Plan -------------- 98
Figure 6.4 Second Alternate Controller Floor Plan ------- 98
Figure 6.5 Microprogram ROM Floor Plan ------------------ 99
Figure 6.6 ROM Array Cells ----------------------------- 101
Figure 6.7 Row Decoder --------------------------------- 105
Figure 6.8 DecodeLSB Cell ------------------------------ 106
Figure 6.9 DecodeEven0 and DecodeEven1 Cells ----------- 107
Figure 6.10 DecodeOdd0 and DecodeOdd1 Cells ------------- 109
Figure 6.11 EvenInverter and OddInverter Cells ---------- 110
Figure 6.12 EvenConnect and OddConnect Cells ------------ 112
Figure 6.13 Two-State Cell ------------------------------ 113
Figure 6.14 Conditional Cells --------------------------- 114
Figure 6.15 COND and CONDtrue Cells --------------------- 116
Figure 6.16 TopCONDfeed and CONDfeed Cells -------------- 117
Figure 6.17 CONDtap, CONDpass, and CONDnopass Cells ----- 119
Figure 6.18 RowBuffer and CONDbuffer Cells -------------- 120
Figure 6.19 LSBbuffer Cell ------------------------------ 121

Figure 6.20 EvenBuffer and OddBuffer Cells -------------- 122
Figure 6.21 ROM Array Cells ----------------------------- 124
Figure 6.22 Row Decode Logic ---------------------------- 126
Figure 6.23 DecodeLSB, DecodeEven0, and DecodeEven1 Logic127
Figure 6.24 NOR Gate ------------------------------------- 128
Figure 6.25 DecodeOdd0 and DecodeOdd1 Logic ------------- 128
Figure 6.26 NAND Gate ----------------------------------- 129
Figure 6.27 EvenInverter and OddInverter Logic ---------- 130
Figure 6.28 Inverter ------------------------------------ 131
Figure 6.29 Two-State Logic ----------------------------- 132
Figure 6.30 COND Logic ---------------------------------- 133
Figure 6.31 CONDtrue Logic ------------------------------ 134
Figure 6.32 RowBuffer and CONDbuffer Logic -------------- 136
Figure 6.33 LSBbuffer Logic ----------------------------- 137
Figure 6.34 EvenBuffer and OddBuffer Logic -------------- 137
Figure 7.1 Taxi Microprogram ROM Design ---------------- 159

Dedicated to my wife, Emilie

I. INTRODUCTION

This thesis encompasses a complete development process. The goal of this process is to produce a Finite State Machine (FSM) controller that can be generated by a silicon compiler. This development supports the silicon compiler project at the Naval Postgraduate School.

A. SILICON COMPILERS

A silicon compiler[1] is a collection of computer programs that translates a high-level description of a Very Large Scale Integrated (VLSI) circuit into a complete layout that can be used to fabricate the circuit. The basic function of a silicon compiler is shown in Figure 1.1. To generate a layout, the silicon compiler produces instances of standard cells and interconnects them as required by the circuit description. The primary advantages of a silicon compiler are speed and reliability of design. A silicon compiler can produce a layout in minutes that would take months for a team of VLSI designers. A good silicon compiler will consistently produce correct layouts. A layout produced by hand may include errors. The disadvantage of using a

[1]For more information on silicon compilers, see Principles of CMOS VLSI Design (Weste and Eshraghian, pp. 250-255, 1985) and VLSI Electronics: Microstructure Science (Einspruch, Vol. 14, pp. 115-138, 1986).

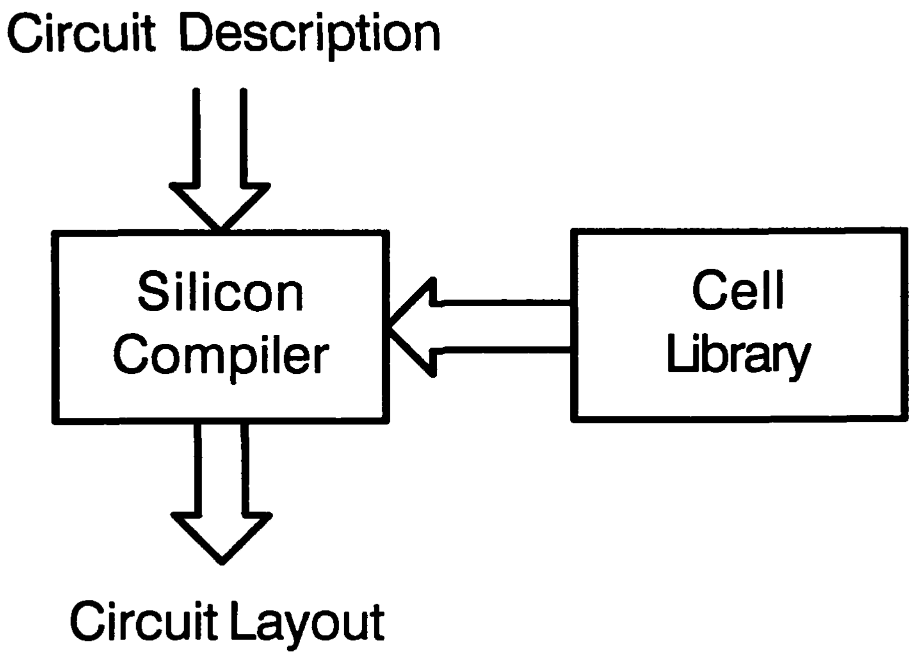


Figure 1.1 Silicon Compiler

silicon compiler is efficiency. A layout produced by a silicon compiler may be larger and slower than a layout produced by a skilled VLSI designer. Although layouts produced by future silicon compilers may be able to surpass the efficiency of layouts produced by human designers, current silicon compilers are most effective for rapid development of custom VLSI circuits that will not have large production runs. For a circuit that will have a large production run, the higher efficiency and smaller size of a human layout will offset the increased design cost that can be amortized over a large number of units. Many military circuits which have small production runs and require high reliability and short development time are ideal for current silicon compilers, provided that performance requirements can be satisfied.

B. MONTEREY SILICON COMPILER (MSC) PROJECT

The Monterey Silicon Compiler (MSC) project at the Naval Postgraduate School is producing a technology independent silicon compiler. Based on the MacPitts[2] silicon compiler produced by MIT Lincoln Laboratory, MSC is written in the LISP[3] programming language. At present, MSC is essentially a modified version of MacPitts. Thus, the nMOS capabilities and architecture of MSC are those of MacPitts. Circuit design specifications processed by MSC are in LISP format. The layout produced by MSC is a Caltech Intermediate Form (CIF)[4] file that can be used to fabricate a circuit at many VLSI foundries. MSC currently supports n-channel Metal Oxide Semiconductor (nMOS) technology. Present development work on MSC involves the addition of Scalable Complementary Metal Oxide Semiconductor (SCMOS) technology. The microprogrammed controller in this thesis supports the SCMOS technology addition to MSC.

[2]Use of the MacPitts silicon compiler is described in Introduction to MacPitts (MIT RVLSI-3, 10 February 1983).

[3]MSC is written in Franz Lisp. For more information on Franz Lisp, see LISPcraft (Wilensky, 1984).

[4]CIF is explained in A Guide to LSI Implementation (Hon and Sequin, pp. 79-123, January 1980).

C. MICROPROGRAMMED CONTROLLER

A microprogrammed controller is an electronic circuit that generates time based control signals for other circuits. The general configuration of a microprogrammed controller is shown in Figure 1.2. It contains a Read Only Memory (ROM) and next address logic. The ROM words contain information for controller output and for next address determination. The status inputs and next address information are used by the next address logic to determine the next ROM address. If each output signal corresponds to one bit of the words in the ROM, the microprogram format is horizontal. If the outputs are produced by decoding a smaller number of

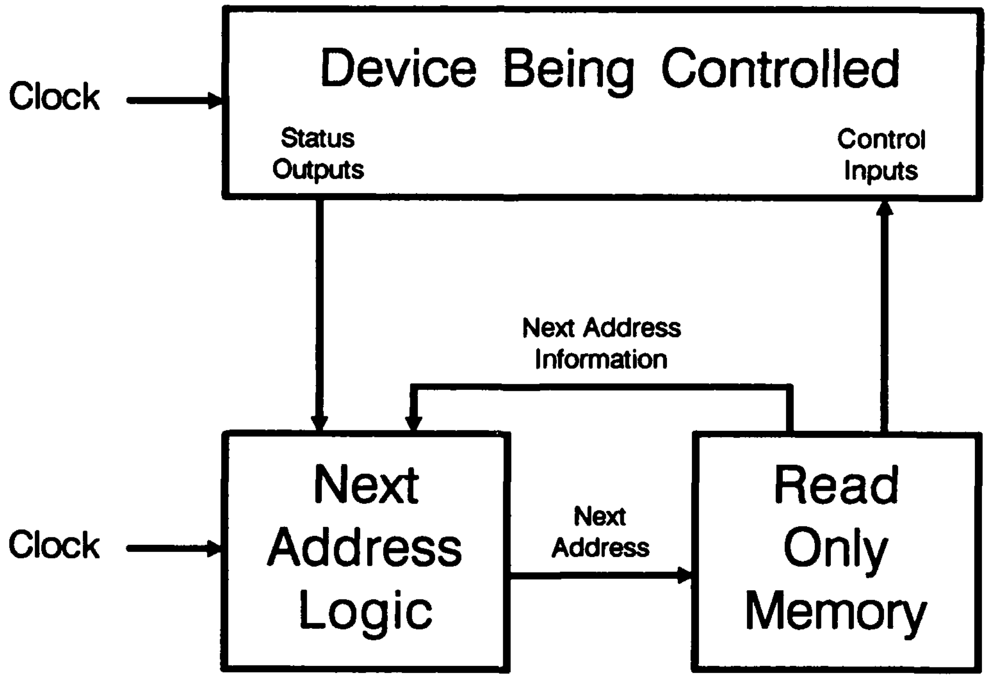


Figure 1.2 Microprogrammed Controller

bits in the ROM words, the microprogram format is vertical. Microprogrammed controllers are used in many applications from traffic lights and microwave ovens to computers. The MSC application of a microprogrammed controller contained in this thesis is similar to the controller in a microprocessor.

D. NAME CONVENTIONS

File name extensions and the file type associated with them are listed in Table 1.1. The extension is the last part of a file name. The library file is an exception to the file name extension convention. This file contains LISP

TABLE 1.1 FILE NAME EXTENSIONS AND FILE TYPES

File Name Extension	File Type
No Extension	Executable File
.c	C Language Source File
.cif	CIF File
.ext	Circuit Extraction File
.l	LISP Language Source File
.mac	MSC Source File
.mag	Magic Layout File
.o	Compiled Object File
.obj	MSC Object File
.sim	Simulation File

formatted statements that are interpreted while MSC is running.

E. CONTENT DESCRIPTION

The organization of this thesis matches a phased development process.[5]

1. Analysis Phase

The analysis phase includes planning, setting goals, and defining requirements. Chapter II starts the analysis phase with an examination of the nMOS layout produced by MSC and development of architectural requirements for the controller. Chapter III includes a description of the programs and functions in the nMOS MSC and software requirements for the controller. Chapter IV completes the analysis phase with a definition of MSC controller goals and a fusion of the requirements developed in Chapters II and III.

2. Design Phase

The design phase starts with trade-off studies in Chapter V. In this chapter, different SCMOS logic structures and controller organizations are evaluated using the goals and requirements of Chapter IV. The logic structure and controller organization selected in Chapter V are used to create the design presented in Chapter VI. The design in

[5]Based on the phased life-cycle model in Software Engineering Concepts (Fairley, pp. 37-42, 1985).

Chapter VI is in two sections. One section is technology independent and one section is for the SCMOS technology.

3. Implementation Phase

The implementation phase is contained in Chapter VII. The first part of implementation involves installation of the nMOS MSC and modification to support the SCMOS technology. The second part of implementation involves layout and test of each cell in the SCMOS section of Chapter VI.

4. Test Phase

The test phase is documented in Chapter VIII. This phase involves integration and test of controllers constructed from the cells generated in Chapter VII. The systems produced are verified against the design in Chapter VII and validated against the requirements in Chapter IV.

5. Maintenance Phase

The maintenance phase of the controller starts with the conclusions presented in Chapter IX. These conclusions involve refinements to the design, methods of installing the controller in MSC, and suggested improvements to the organization of MSC. Additional research in support of MSC is included in these conclusions.

II. MSC TARGET ARCHITECTURE

Understanding the physical and functional properties of the MSC target architecture is crucial to the formulation of requirements for the microprogrammed controller that is the subject of this thesis. MSC produces layouts for complete nMOS VLSI circuits that may include one or more finite state machine controllers.

A. PHYSICAL LAYOUT

An MSC layout includes pads, a control logic array, a data path, a flags area, and distribution structures for signals and power. Figure 2.1 shows the floor plan of an MSC nMOS VLSI layout. The perimeter of the layout is a pad frame with pads on three sides. The pads are numbered sequentially in a clockwise direction starting at the left pad on the top edge. Inside the pad frame is a frame for drain power (Vdd) and ground (GND) distribution. In between the pad frame and the Vdd and GND frame is a routing channel for signals to and from the pads. Inside the Vdd and GND frame are the data path, flags area, and control logic array. The data path and control logic array are separated by horizontal clock and power buses. The flags area is to the right of the data path. The pads are used for connections between the VLSI circuit and the package that encloses it. Word size data processing is accomplished in the data path and

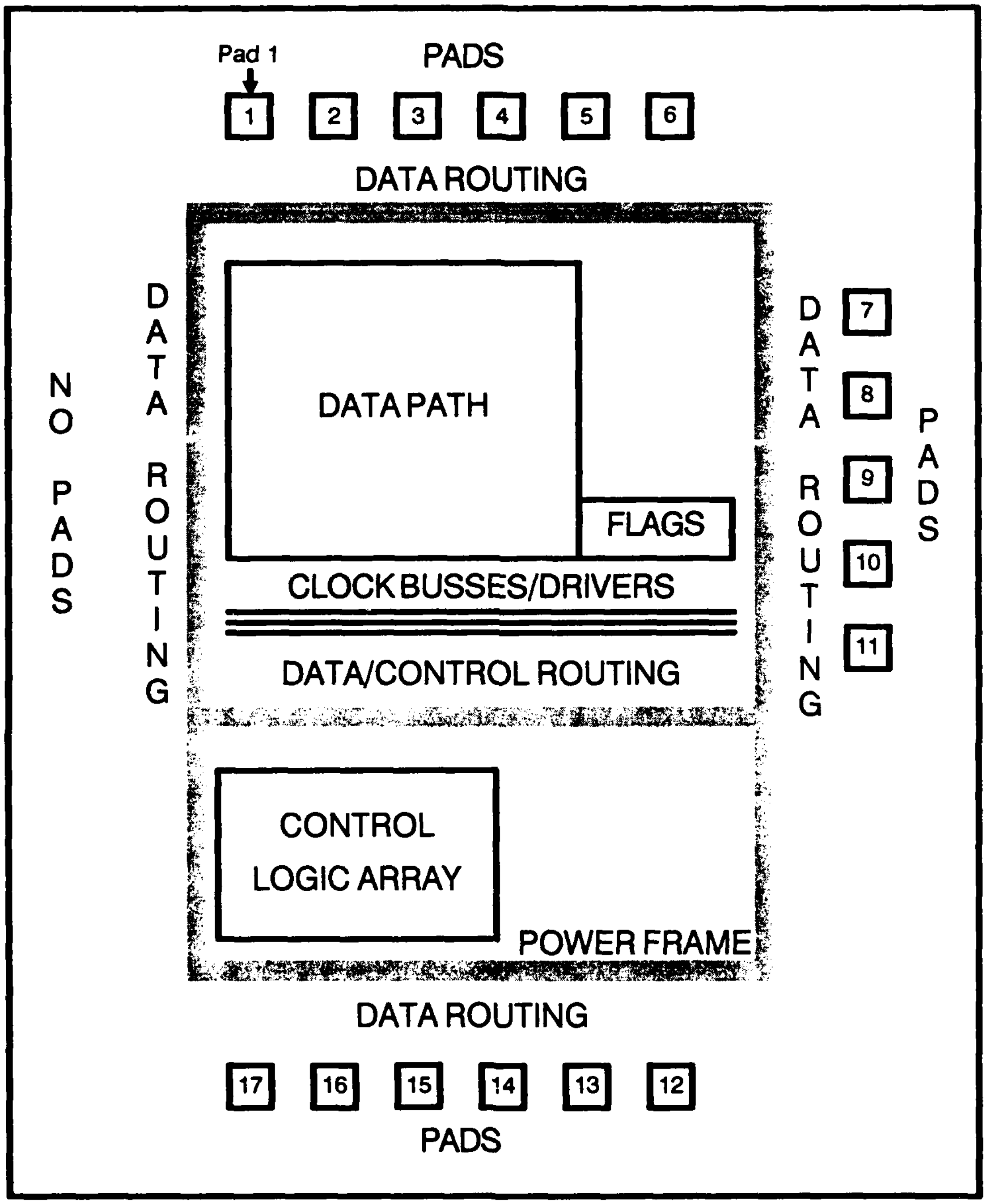


Figure 2.1 MSC Floor Plan

single bit processing is accomplished in the control logic array. Word size data storage is performed in the data path and single bit storage is performed in the flags area.

1. Data Path

The data path in an MSC layout is the area that contains most of the structures used for data processing and data storage. It is organized horizontally in bit slices and vertically in units. This organization is shown in Figure 2.2. Each horizontal bit slice contains all of the processing elements for one bit of word size data. Each vertical unit contains similar elements that perform one function on all bits in word size data structures. There are five types of units: register, output port, internal port, bit, and organelle. Register units, shown in Figure

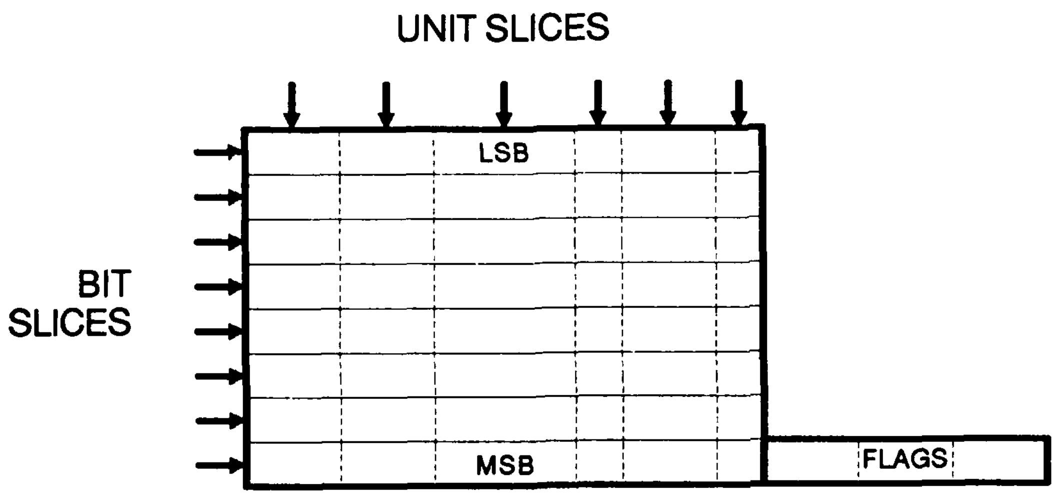


Figure 2.2 Data Path Organization

2.3[1], are used for data storage. Output port units are used to generate multiple bit output words for pads. Internal port units are used to generate multiple bit words for use inside the data path. Output port units and internal port units have the same internal structure as shown in Figure 2.4. Bit units, shown in Figure 2.5, are used to extract groups of bits for the control logic array from multiple bit words in the data path. Organelle units, shown in Figure 2.6, generate single bit outputs of multiple bit logic or arithmetic functions. Each unit may include one or two multiplexers to select input sources. Each unit input is called an argument. Each argument source is called an operand. For operands that are constant, the operand inputs are hard wired to ground or power to force constant logical zero or one inputs. For arguments with single operands, a degenerate multiplexer that is a direct connection from the operand to the argument is used. These alternate multiplexer configurations are shown in Figure 2.7. Operand selection for the multiplexers in each unit is controlled by a bus of multiplexer control lines from the control logic array. Each multiplexer control line enables one operand for each argument in the unit. Data path internal buses,

[1]The units in Figures 2.3 - 2.6 are all shown with two input multiplexers for each argument. The actual number of inputs to each multiplexer in a unit is determined by the number of unique sources and constant values associated with each argument.

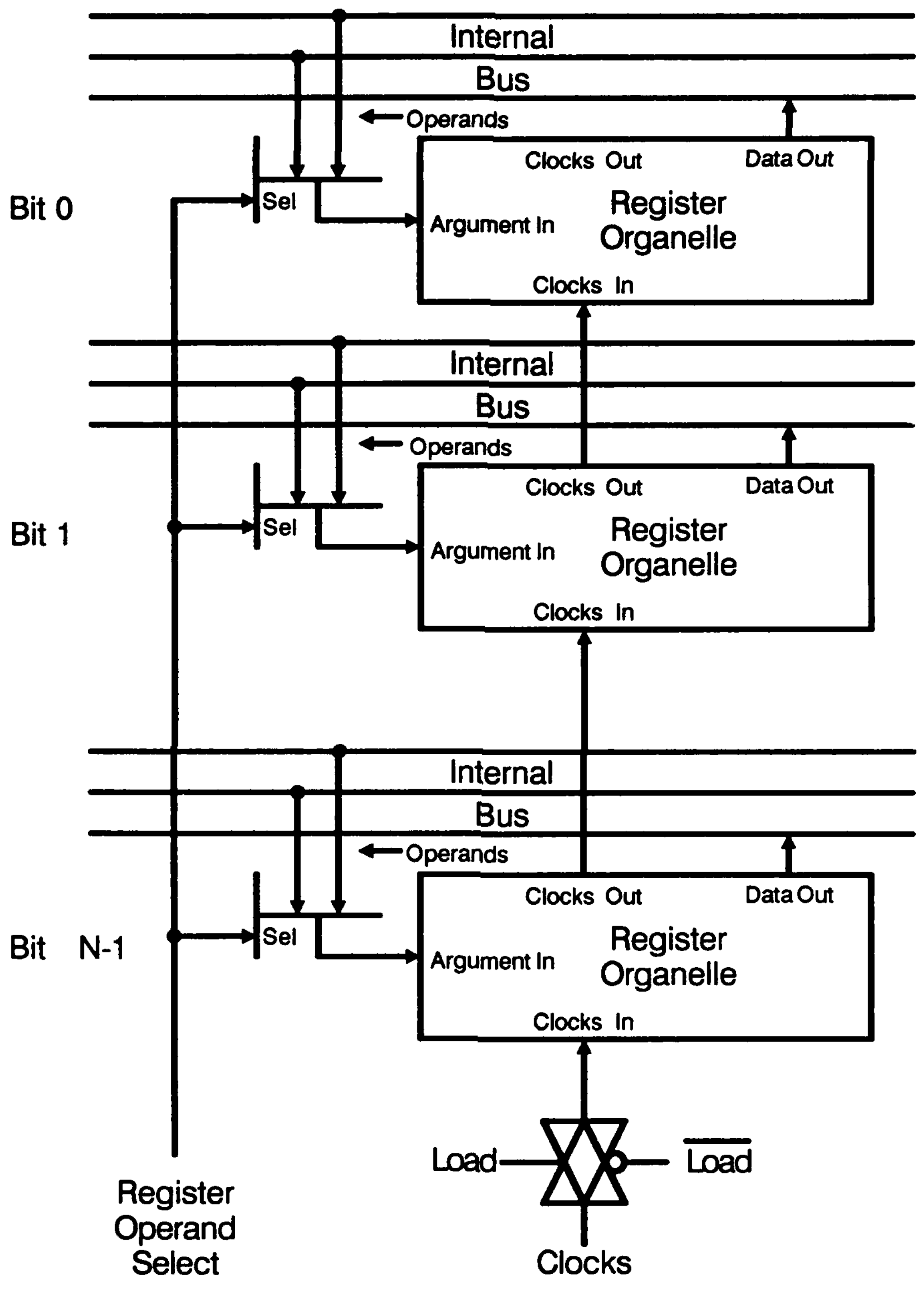


Figure 2.3 Register Unit

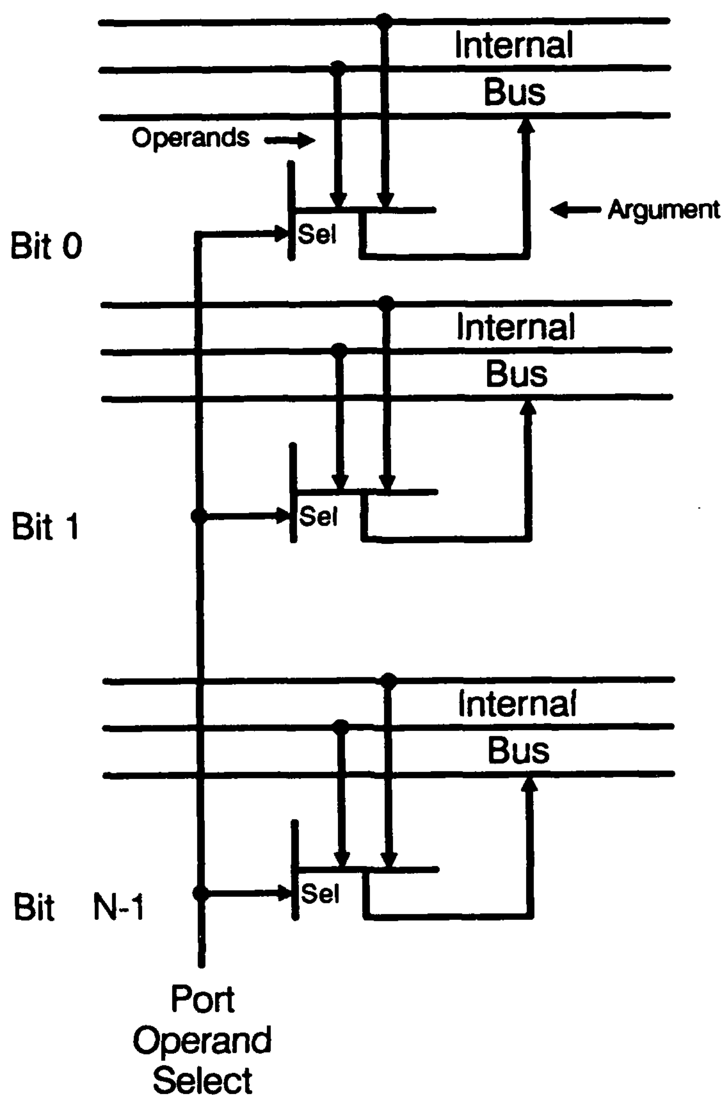


Figure 2.4 Output Port and Internal Port Units

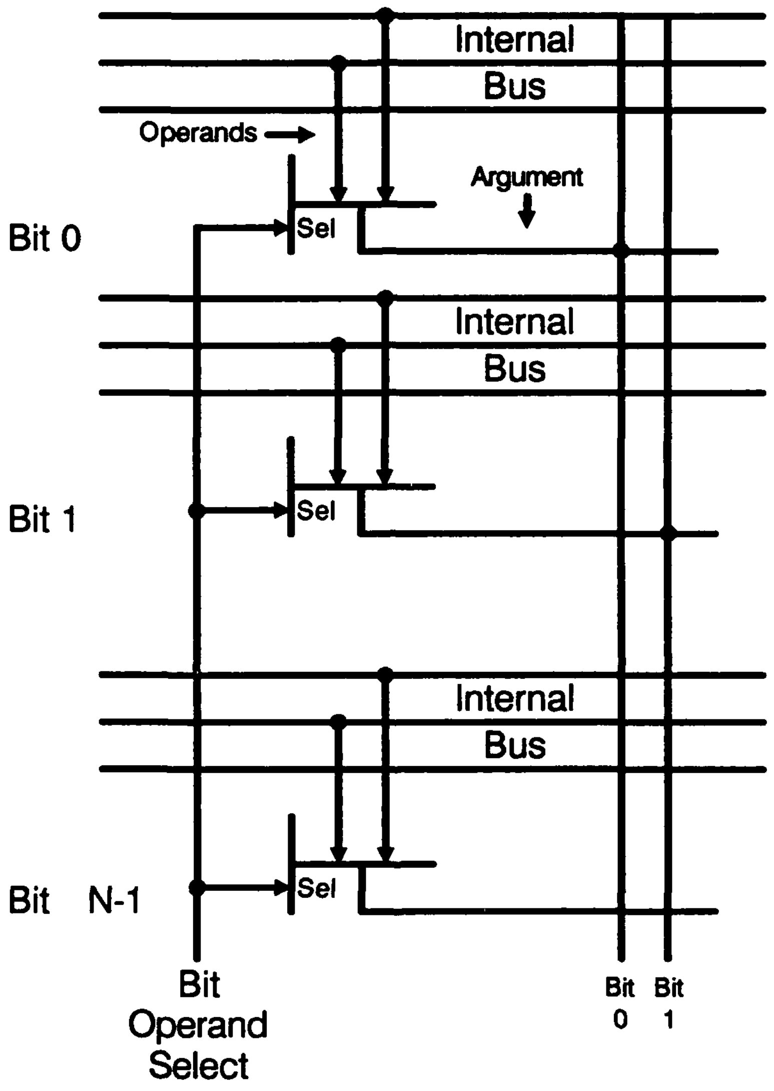


Figure 2.5 Bit Unit

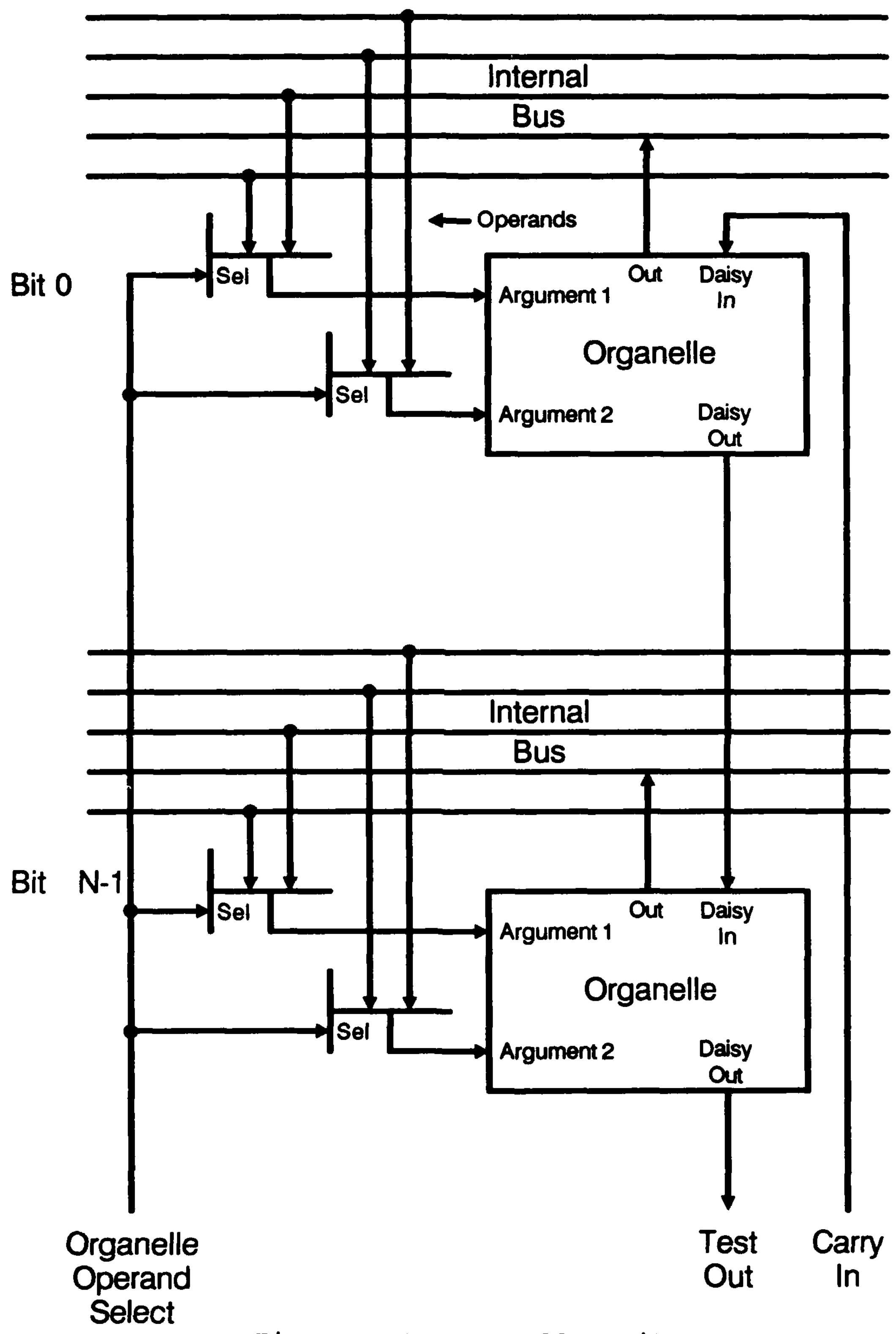
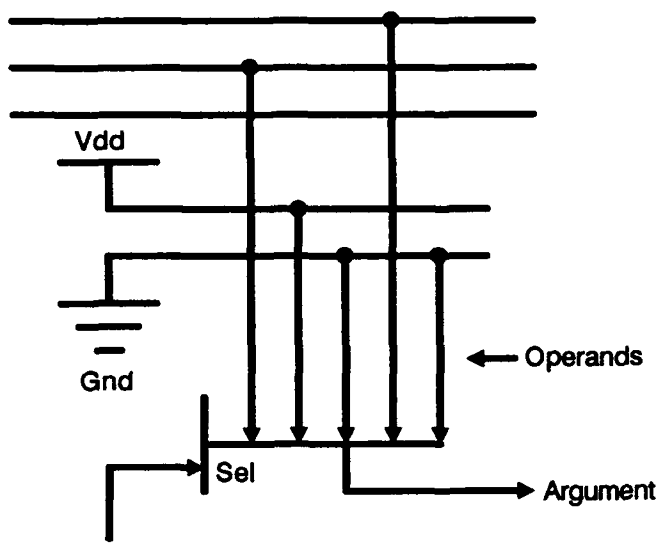
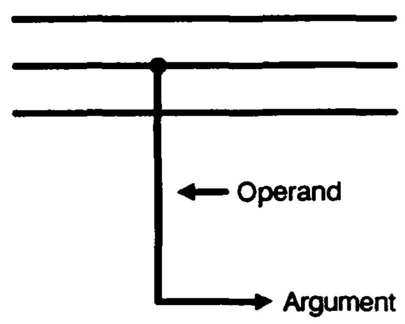


Figure 2.6 Organelle Unit





Multiplexer with Constant Inputs

Multiplexer with Single Input

Figure 2.7 Unit Multiplexer Options

located at the top of each bit slice, connect register, output port, internal port, bit, and organelle units. The bit slices are stacked to form the data path.

2. Control Logic Array

The control logic array generates control and data signals that are sent to the data path and pads. All single bit processing is performed in the control logic array. The inputs to the array are signals from organelles, registers, flags, and input pads. The outputs are control signals for the registers and multiplexers or data signals that are used for setting flags, internal signals, or output pads. The current nMOS version of MSC uses a Weinberger[2] logic array. Unlike the Programmable Logic Array (PLA), the Weinberger array is composed exclusively of NOR gates. For example, an exclusive or (XOR) of two single bit signals is converted to an equivalent combination of NOR gates in the control logic array. The inputs and outputs both run in vertical columns. Ground distribution is also done in vertical columns. The horizontal tracks are used to determine the relationship between the inputs and outputs.

B. FINITE STATE MACHINE (FSM) ARCHITECTURE

A finite state machine (FSM) is a clocked circuit that "remembers information about its past inputs"(Langdon, 1982, p 522). It is composed of combinational logic and clocked memory elements that store state variables. "The state is the totality of the values of all bits ... in storage."(Langdon, 1982, p 522) The relationship between the inputs and outputs is determined by the current state of the machine. During each clock cycle, new values of the state variables are loaded into the memory elements and the machine transitions to a new state. The new state variable values are the next state outputs from the combinational logic. These next state outputs are based on the current inputs and the old state values stored in the memory

[2]For a description of a Weinberger array see "Large Scale Integration of MOS Complex Logic: A Layout Method"(Weinberger, 1967).

elements. There are two types of FSMs, the Moore FSM and the Mealy FSM.

1. Moore FSM

The Moore FSM shown in Figure 2.8 is the simplest form of FSM. It consists of storage elements for the state variables and combinational logic used to determine the next values of the state variables based on the current values of the state variables and the input signals. The outputs are functions of only the state variables. The input signals

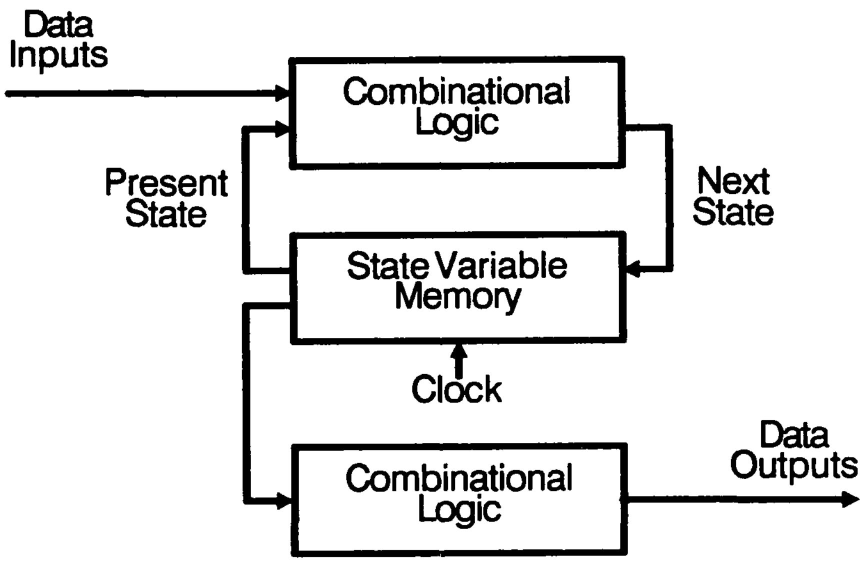


Figure 2.8 Moore Finite State Machine

affect the next state but not the current outputs. As a result, the outputs only change during state transition.

2. Mealy FSM

The other type of FSM, the Mealy FSM, is shown in Figure 2.9. In a Mealy FSM, the outputs are functions of the state variables and the input signals. The output signals may change in response to changes in the input signals.

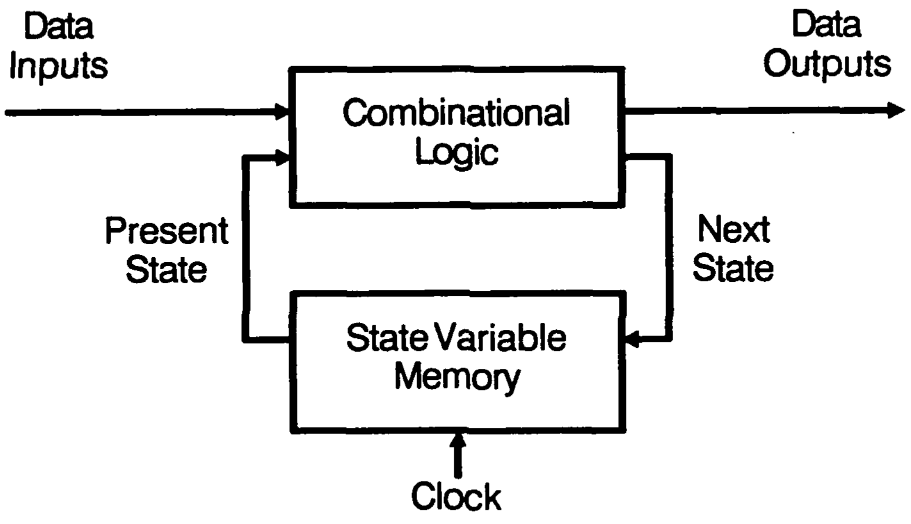


Figure 2.9 Mealy Finite State Machine

3. Moore FSM Timing

The timing relationships between the input and output signals of a Moore FSM are shown in Figure 2.10. The inputs must be stable prior to state transition for a length of time equal to the sum of the setup time for the registers and the maximum delay time for the combinational logic. The

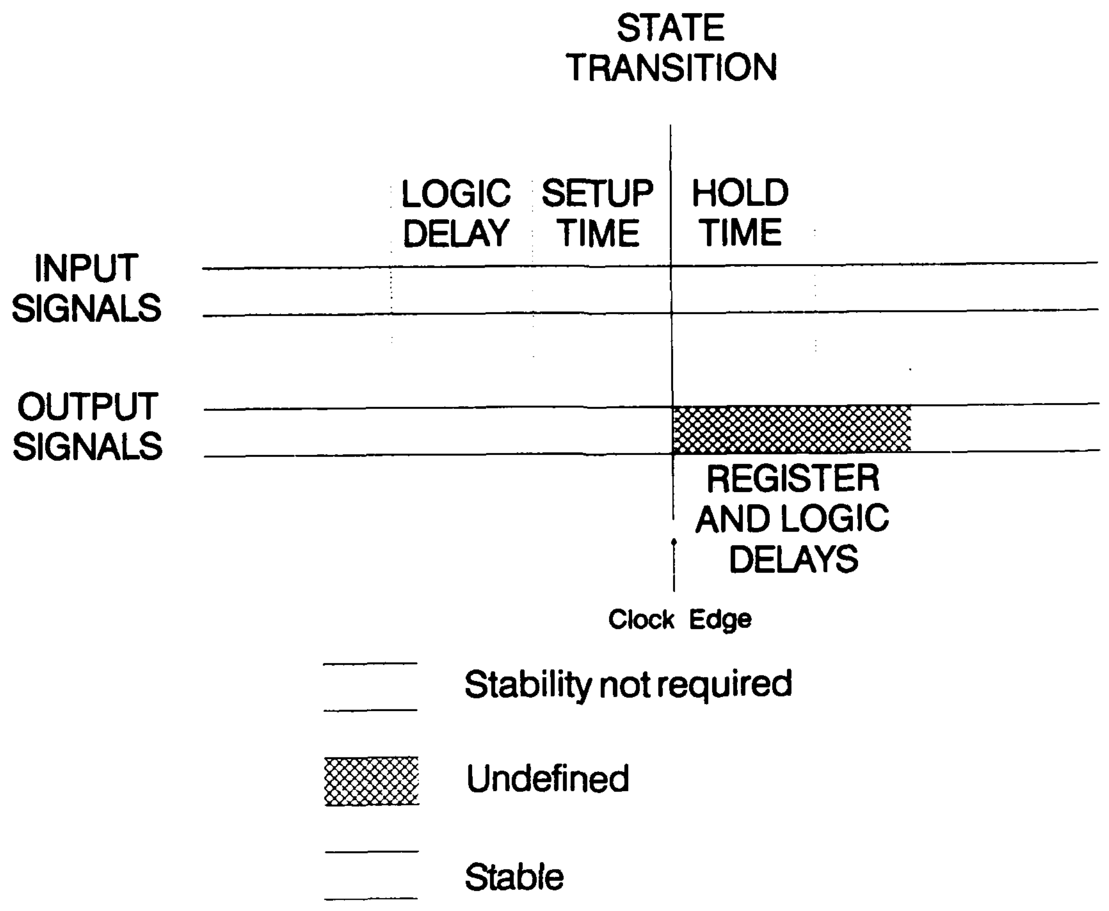


Figure 2.10 Moore FSM Timing

inputs must remain stable after state transition (clock edge) long enough to satisfy register hold time requirements. The outputs of a Moore FSM become undetermined at state transition and remain undetermined for the duration of the register and combinational logic delays. After the outputs have stabilized, they remain stable until the next state transition.

4. <u>Mealy FSM Timing</u>

The timing relationships between the input signals and output signals that depend only on state variables of the Mealy FSM are the same as the relationships between the input and output signals of the Moore FSM shown in Figure 2.10. The timing relationships between the input signals and output signals that are dependent on state variables and input signals of a Mealy FSM are shown in Figure 2.11. Like the Moore FSM, the inputs to the Mealy FSM must be stable prior to state transition for a length of time equal to the sum of the setup time for the registers and the maximum delay time for the combinational logic. The inputs must remain stable after state transition (clock edge) long enough to satisfy register hold time requirements. The outputs of a Mealy FSM become undetermined at state transition and remain undetermined for the duration of the register and combinational logic delays. After the outputs that depend only on state variables have stabilized, they remain stable until the next state transition. The outputs that also depend on current inputs are stable only when the inputs they depend on have been stable long enough for the outputs of the combinational logic to stabilize.

5. <u>State Transition Diagram</u>

An FSM has a fixed number of states. It can transition to a subset of these states based on the current state. In a state transition diagram, this is represented as a

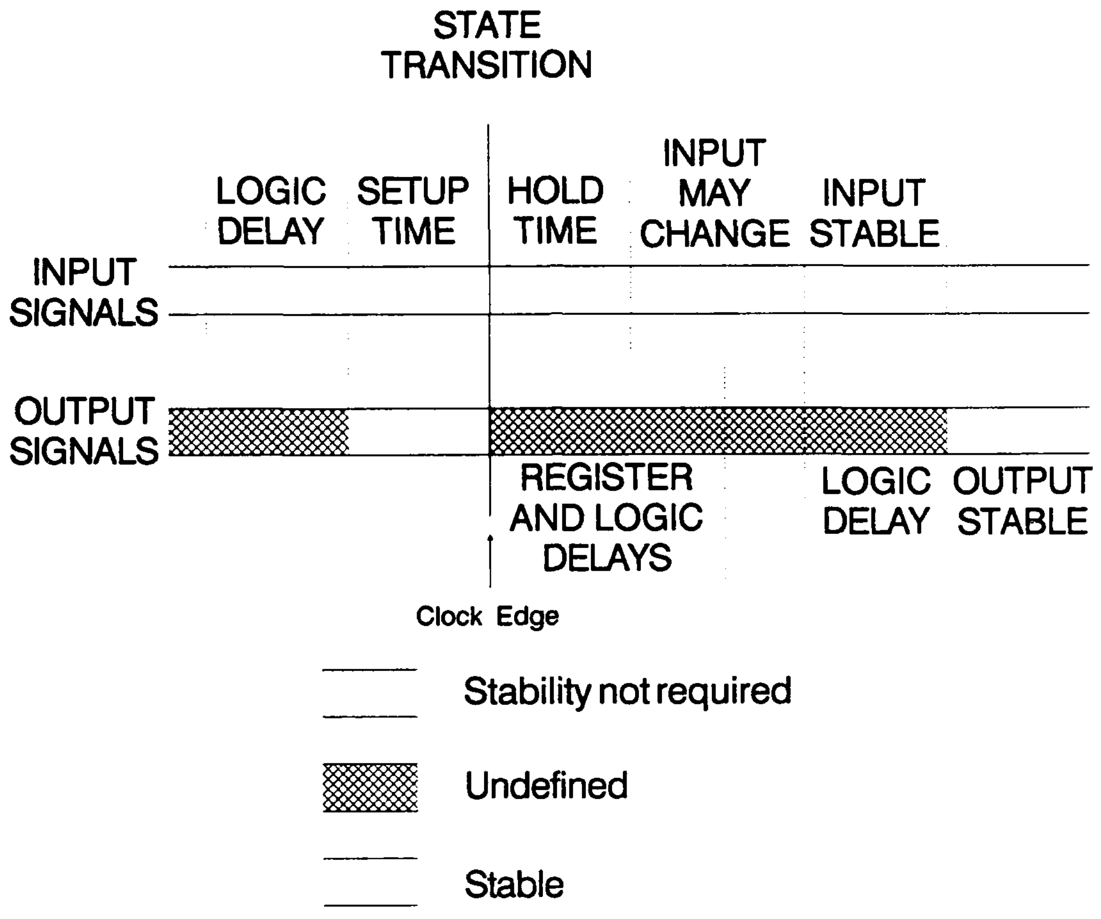


Figure 2.11 Mealy FSM Timing

graph with nodes for the states and directed branches for available state transitions. The choice of which branch to take from a node is based on inputs to the FSM.

6. <u>Moore FSM State Transition Diagram</u>

In the state transition diagram for a Moore FSM, the outputs are associated with the nodes. An example of a Moore FSM state transition diagram for a J-K flip-flop is shown in Figure 2.12. The J-K flip-flop is a simple FSM with a single binary state variable. This single state variable is also the output of the J-K flip-flop. There are two states A and B. When the FSM is in state A, the output is zero. When the FSM is in state B, the output is one. These states are drawn as circular nodes. Each node has branches leaving it that return to the same node or transition to the other node. The branches represent the values of the input signals during the next state transition. If the current state is A and J is one, the next state will be B. This is the lower branch. If the current state is A and J is zero, the next state will be A. This is the left branch that returns to the same state. If both J and K are

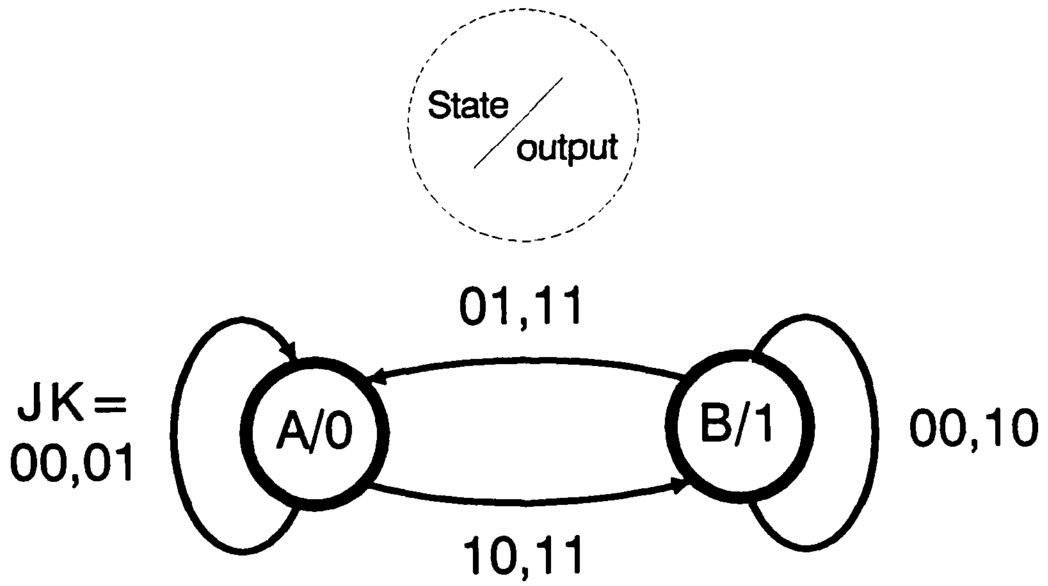


Figure 2.12 Moore FSM State Transition Diagram

zero, the state does not change. If both J and K are one, the state will alternate between A and B every clock cycle.

7. Mealy FSM State Transition Diagram

The state transition diagram for a Mealy FSM is similar to the state transition diagram for a Moore FSM. The only difference is that in the Mealy FSM state transition diagram, the outputs are associated with the branches. An example of a Mealy FSM state transition diagram for a transition detector circuit is shown in Figure 2.13. A transition detector compares the current input with the input at the last state transition. If they are the same, no transition has occurred and a zero is output. If the current and previous inputs are different, a transition has occurred and a one is output. The state transition diagram also has two states A and B. State A corresponds to a zero input at the

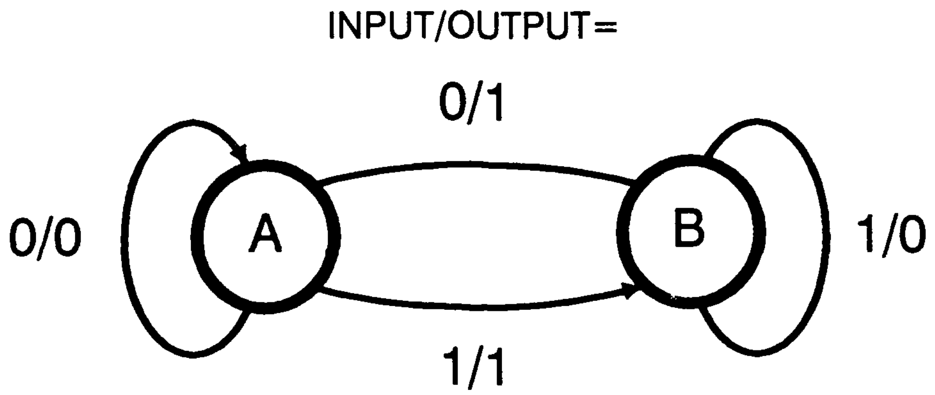


Figure 2.13 Mealy FSM State Transition Diagram

last state transition and state B corresponds to a one input at the last state transition. The left label on each branch is the current input and the right label on each branch is the current output.

C. MSC FSM STRUCTURE

The layout produced by MSC may contain one or more modified Mealy FSMs. The structure of these FSMs is shown in Figure 2.14. The only state variable used by each of these FSMs is a state number. The state number is an unsigned integer in the range of 0 to $2^W - 1$, where W is the data path word size. The maximum number of states for each FSM is 2^W. Each FSM may include a fixed depth stack and an incrementer for state numbers. The selection control of the multiplexer and the push/pop controls of the optional state number stack are determined by the MSC design specification of the current state. If the current state description contains a call statement, the multiplexer selects a call state number from the combinational logic and a return state from the incrementer is pushed onto the stack. In this way, the FSM transitions to the state specified in the call statement and the number of the state following the call statement is pushed onto the stack for use by a return statement. If the current state description contains a return statement, the multiplexer selects the return state number on the top of the stack and the stack is popped. This causes the FSM to

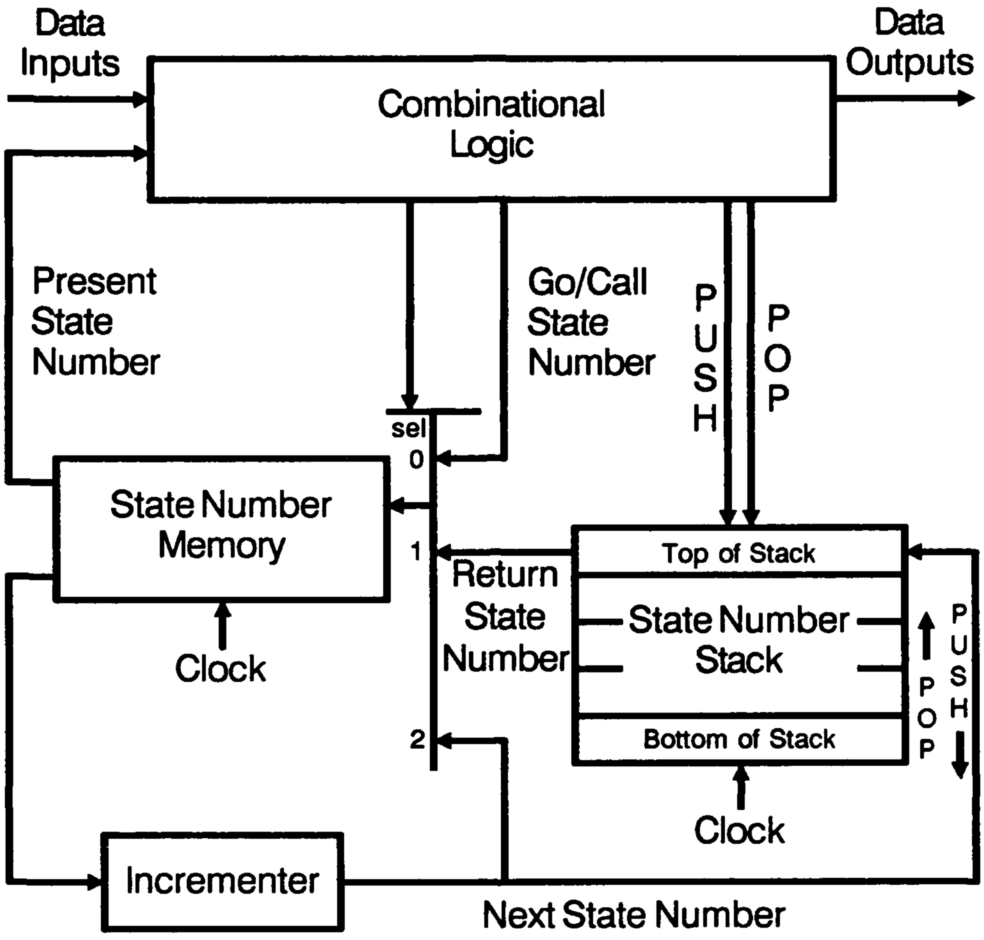


Figure 2.14 MSC Finite State Machine

transition to the state following the state with the most recently executed call statement. The calls and returns may be nested so that a subroutine may call another subroutine, or a subroutine may call itself recursively. There is no checking for stack overflow or underflow on subroutine calls and returns. The stack depth specified in the MSC design specification for a process must be large enough to accommodate the deepest nesting of subroutines expected in the process. This depth may be difficult to estimate for recursive subroutines. If the current state description contains a go statement, the multiplexer selects the go state number from the combinational logic. If the current state (other than the last state in a process) does not include a call, return, or go statement, the multiplexer selects the next state number from the incrementer. The last state in a process contains an implicit go statement to the first state in the process. This implicit go statement is overridden by any go, call, or return statements in the last state of a process. The incrementer is used to determine the next sequential state. If a design has more than one process, each of the processes is controlled by a separate FSM.

D. MULTIPLE FSM INTERACTION

There is a separate FSM for each process. The separate FSMs simultaneously control the common data processing and storage elements as shown in Figure 2.15. These processes communicate with each other by registers, flags, signals, and internal ports. An MSC register is a multiple bit data structure that continuously stores a value and has the same number of bits as the data path word size. A flag is a one bit data structure that continuously stores a value. A

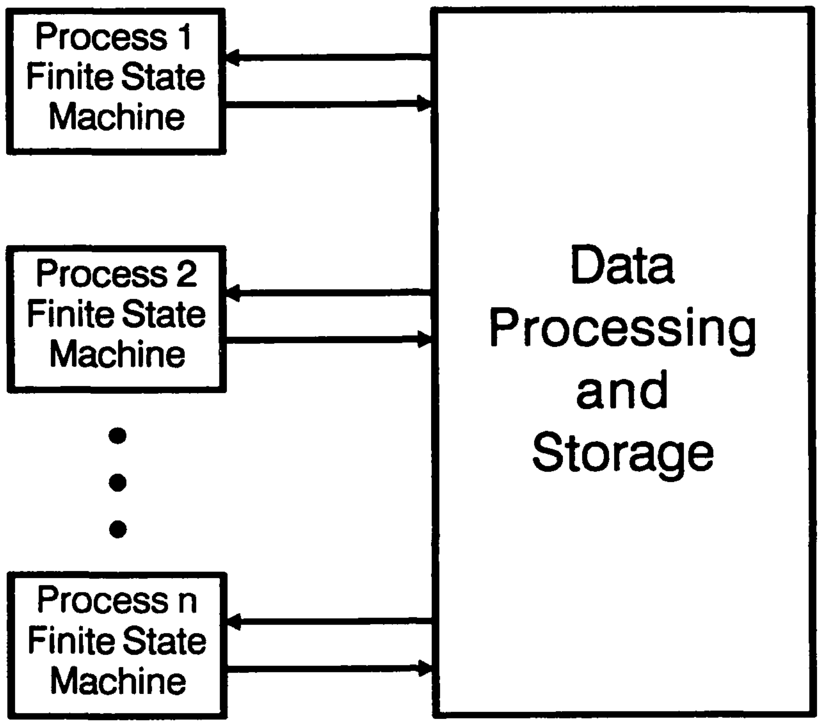

Process 1 Finite State Machine
Process 2 Finite State Machine
Process n Finite State Machine
Data Processing and Storage

Figure 2.15 Multiple FSM Interaction

signal is a one bit data structure that only has a value during the state in which its value is set. An internal port is a multiple bit data structure that has the same number of bits as the data path word size and only has a value during the state in which its value is set. Each register, flag, signal, or internal port is controlled by a single FSM controller.

1. <u>nMOS Multiplexer Wired AND Logic</u>

The multiplexers used in the nMOS version of MSC produce a logical AND of their input values when more than one input is selected simultaneously. This would happen when a par statement or a single alternative of a cond statement for a state in a process or an always statement in the design specification assigns multiple values to a register or port. The following three segments of a design specification attempt to assign the value of b to a and the value of c to a simultaneously:

```
(always (setq a b) (setq a c))
(par (setq a b) (setq a c))
(cond (t (setq a b) (setq a c)))
```

The action of the wired AND produced by the nMOS transmission gate multiplexers used in MSC is shown in Figure 2.16[3]. This shows the outputs of two inverters connected to the transmission gate multiplexer in the center of the figure. The output of the multiplexer is labeled Out. The selection controls of the multiplexer are labeled Sel 1 and Sel 2. The inputs to the inverters are labeled In 1 and In 2. The inputs to the multiplexer are the complements of the inputs to the inverters or (NOT In 1) and (NOT In 2). These input signals to the multiplexer are low when the inputs to the corresponding inverters are high causing the enhancement mode pull down transistors to conduct and pull the outputs down to GND. The input signals to the multiplexer are high when the inputs to the corresponding

[3]For a description of the symbols used in this schematic and all subsequent schematics, see Figure 1.11(b) on page 12 of The Design and Analysis of VLSI Circuits (Glasser and Dobberpuhl, p. 12, 1985).

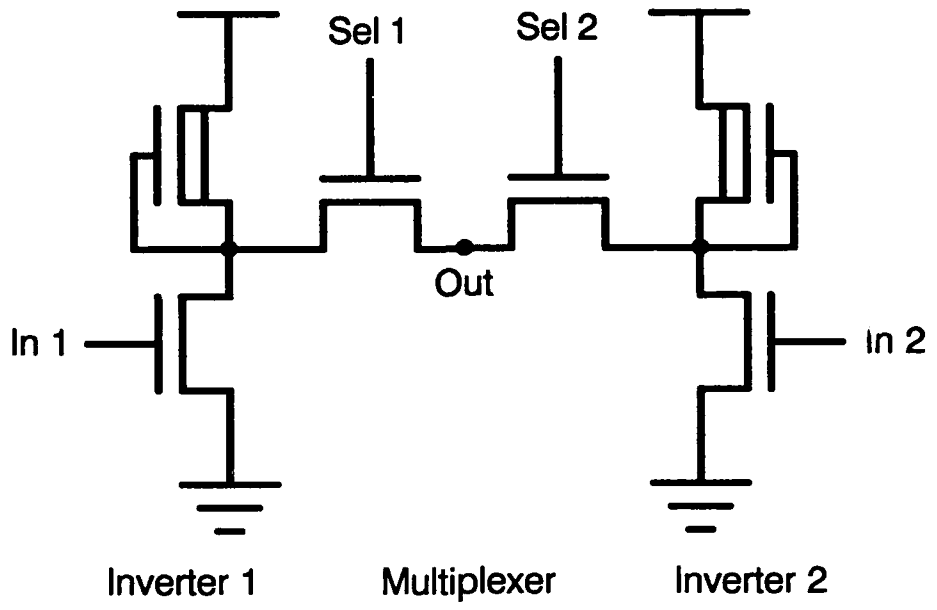


Figure 2.16 Multiplexer Wired AND Logic

inverters are low causing the enhancement mode pull down transistors to be cut off which allows the depletion mode pull up transistors to pull the outputs up to Vdd. If only one of the Sel signals is high, the signal at Out is the normal output of the corresponding inverter. If both of the Sel signals are high, the outputs of both inverters are connected to the output of the multiplexer. This also connects the outputs of the inverters to each other allowing either pull down transistor to pull down all outputs. This turns the entire circuit into a NOR gate where the output is (NOT (In 1 OR In 2)). This can be converted to ((NOT In 1) AND

(NOT In 2)) by DeMorgan's Laws[4]. This is the logical AND of the inputs to the multiplexer. This nMOS wired AND in the multiplexer includes the side effect that the outputs of the inverters are the AND of the normal outputs throughout the circuit. If one of the inputs to the multiplexer is a register, the value stored in the register is forced to the logical AND of the inputs to the multiplexer. This reverses the normal signal flow from the output of a register to the input of a multiplexer. This wired AND can destroy a circuit if one of the inputs is a constant high (Vdd) and the other is a constant low (GND). If the multiplexer selects both these inputs simultaneously, there is a direct path from Vdd to GND that would probably burn out the multiplexer. The current nMOS MSC cannot generate a multiplexer that is controlled by two processes. The internal functional simulation of MSC generates an error if any of the multiplexers in a design specification can have more than one input selected at a time. The ESIM switch level simulations of nMOS MSC designs with multiple simultaneous inputs to multiplexers verify the operation of the wired AND logic.

2. SCMOS Multiplexer Logic

The SCMOS multiplexer and the nMOS multiplexer should have the same function. SCMOS circuits use low

[4]For a statement of DeMorgan's Laws and logical tautologies, see Introduction to LOGIC(Suppes, p. 34, 1957).

impedance pull up and pull down structures. This precludes the use of wired logic. The SCMOS multiplexer cannot duplicate the operation of the nMOS multiplexer for multiple simultaneous input selections. Since multiple simultaneous input selections to multiplexers generate errors for the internal functional simulation of MSC and cause differences between SCMOS and nMOS circuits, multiple simultaneous input selections to multiplexers should not be supported by MSC. The SCMOS multiplexer should function as a true multiplexer with no side effects.

E. ARCHITECTURAL REQUIREMENTS

The MSC target architecture implies required characteristics of a controller. The controller must be composed of multiple Mealy FSMs. Each FSM must have a single state variable that is the state number. Each FSM must be able to include an incrementer and subroutine stack. Each unit in the data path must be controlled by a single FSM or always using true multiplexers that produce no side effects. These requirements will be used to evaluate the suitability of controller designs.

III. MSC INTERNAL STRUCTURE

The internal structure of MSC is based on the MacPitts silicon compiler. Since MSC is currently a modified version of MacPitts, the internal structures of the two silicon compilers are identical. Analysis of the MSC structure is used to generate requirements for the controller and to eliminate design alternatives that are not consistent with this structure. MSC is a collection of programs that extract information from a design specification to produce a functional simulation of the specification or a Caltech Intermediate Form (CIF)[1] file describing a VLSI circuit that implements the specification.

A. DATA PRIMITIVES

There are four data primitives defined for MSC: signal, flag, port, and register. These are shown in Table 3.1. The MSC data primitives correspond to internal nodes in a VLSI circuit. Signals and flags represent single bit nodes and ports and registers represent multibit nodes. The number of bits in ports and registers is determined by the word-size specified in a design specification. All ports and registers in a MSC VLSI circuit have the same number of

[1]For a description of the format and use of CIF files, see the CIF primer in A Guide to LSI Implementation(Hon, 1980, pp. 79-123).

bits. Signal and port nodes do not have memory while flag and register nodes do have memory. Nodes that do not have memory have valid data only during a state that assigns a value to them. Nodes with memory retain the values that were last assigned to them until the next state transition. If no new value is assigned to a node with memory during a state transition, the node retains its previous value. Signals and ports may be internal or external. External nodes are associated with pads in a MSC VLSI circuit and internal nodes are not associated with pads.

TABLE 3.1 DATA PRIMITIVES

	Single Bit	Multi-Bit
No Memory	signal	port
Memory	flag	register

B. DATA STRUCTURES

Understanding a program's data structures is a prerequisite to understanding its operation. The data structures of a computer program limit the information that it can input, manipulate, and output. MSC uses a combination of program defined and defstruct[2] data structures. Program defined data structures are those that are not explicitly stated in the program source code. The program defined data structures are implied by the functions that create instances of these data structures or access information in them. All of these functions are defined in the MSC source code. Defstruct data structures are explicitly stated using the defstruct macro. All functions to create instances of defstruct data structures or access information in them are automatically generated by the defstruct macro. Both of these methods use Lisp lists to store data. A design specification[3] is a text file that contains a functional description of a VLSI circuit to be compiled by MSC. Both the design specification and the library[4] program are read by MSC at run time. The definition and eval data structures in the design specification are written by the user. The definition and eval data structures in the library program are part of the MSC program. The functions that extract

[2]Defstruct is a Lisp define structure macro in the lincoln.l program. The defstruct macro defines a Lisp data structure by creating all of the functions required to generate an instance of the structure and to access all parts of the structure.(Malagon-Fajar, 1986, pp. 87-95)

[3]An MSC design specification is a text file whose name ends in .mac. The data structures used an MSC design specification are described in An Introduction to MacPitts(MIT RVLSI-3, 1983).

[4]The organization and use of the library program is discussed in Paragraph III.D.7.

information from these structures are located in the prepass.l and general.l programs. All other data structures are created and accessed by functions generated by the defstruct macro. The layout structure for an item composed of points, rectangles, symbol names, and operators and the hierarchical symbol layout data structure are defined in the program L5.l. The symbol data structure is created and manipulated by functions in the L5.l program. The object structure in an .obj file, the extraction structure used to produce it, and other internal data structures are defined in the program defstructs.l.

C. CONTROL AND DATA FLOW

The control flow of MSC is shown in Figure 3.1. The MSC compiler is controlled by the top level function macpitts-compiler in prepass.l. At the start of a compilation, command line options are processed by functions in prepass.l. If a functional simulation is requested, the design specification is interpreted by functions in the program interpret.l. The design specification file and library file are read by functions in prepass.l. Definition extraction, eval evaluation, macro expansion, and constant substitution are performed by functions in prepass.l and general.l on the information in the design specification file and library. The compiled organelles.o program is loaded by an eval structure in library. If an .obj file is requested,

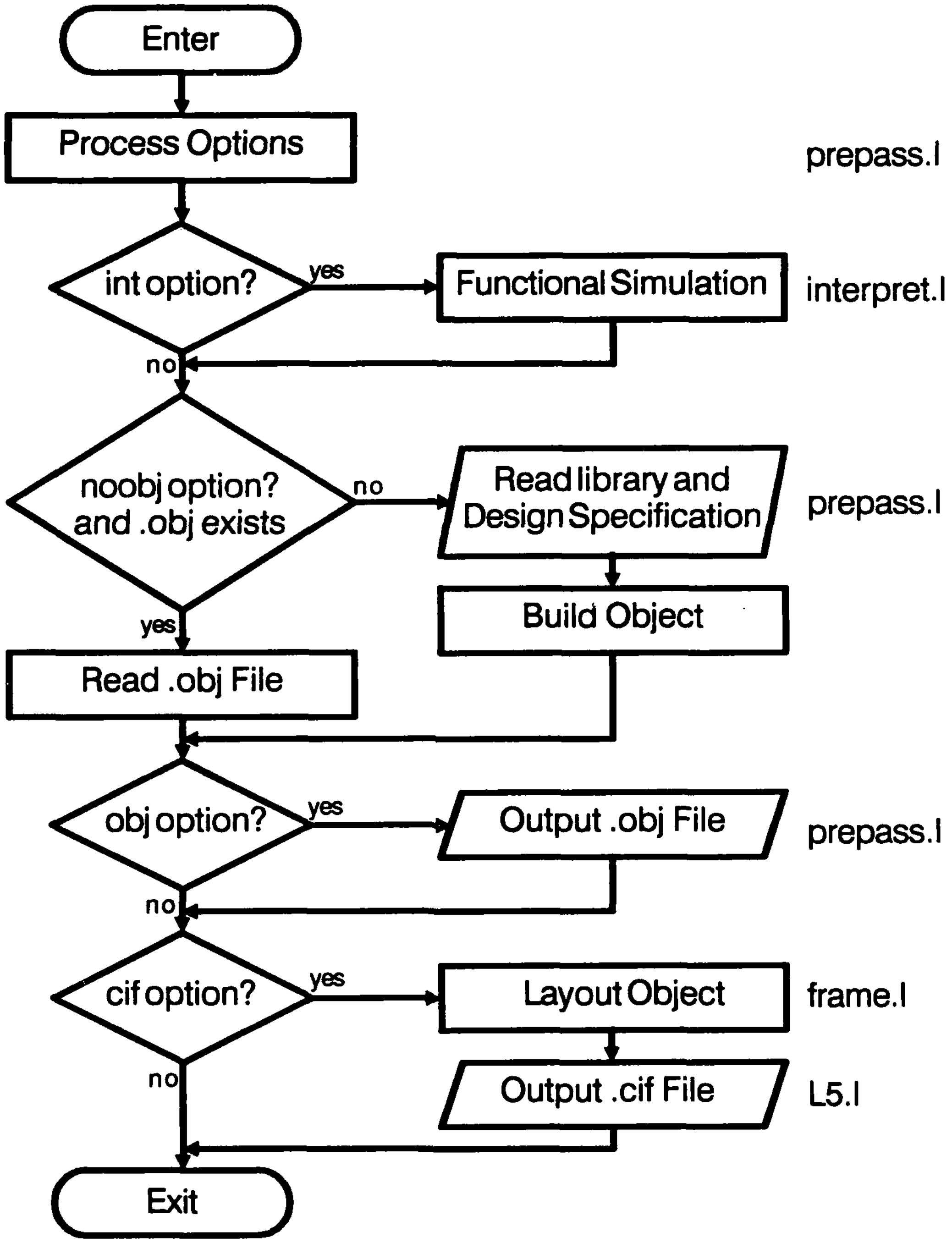


Figure 3.1 MSC Control Flow

the data path, flags, control, and pad information for the object structure in the file is extracted and reduced by functions in extract.l. If an .obj file is not requested and an .obj file exists, the .obj file is read into an object structure by the get-object function in prepass.l. If CIF output is requested, the object structure is converted into an item by the layout-object function in the frame.l program. The item is output in CIF format by the cifout function in L5.l.

D. PROGRAM COMPONENTS

MSC is generated under the control of a Makefile[5] by loading compiled programs into Franz Lisp and using the dumplisp function to save the resulting Lisp environment in an executable file. One program source, c-routines.c, is written in the C programming language. All other program sources are written in Lisp.

1. Program lincoln.l

Program lincoln.l contains MIT Lincoln Laboratory general purpose extensions to the Franz Lisp environment. Functions to add use of the vi editor from inside the Franz Lisp environment, access to the UNIX[6] curses[7] terminal

[5]For a description of the make utility, see "Made to Order"(Shelly, 1986, pp. 128-131) or Make - A Program for Maintaining Computer Programs(Feldman, 1978).

[6]UNIX is a Trademark of Bell Laboratories.

screen updating and cursor movement optimization package, APL[8] like operators for lists, set functions, list selection functions, numeric functions, and program beautification macros to Franz Lisp and the Liszt compiler. The most important component of lincoln.l is the defstruct (define structure) macro. The defstruct macro is used to define the majority of the data structures used in MSC.(Crouch, 1984)

2. Program c-routines.c

The program c-routines.c is the only component of MSC that is written in the C programming language. The functions in this program provide Franz Lisp bindings for the routines in the UNIX curses library. The compiled version of this program, c-routines.o, is loaded by lincoln.l. The only component of MSC that uses these routines is interpret.l. Interpret.l uses the routines in c-routines.c to update the terminal display during functional simulation of a design specification.

---

[7]Curses is a library of terminal independent screen updating functions.(UNIX Programmer's Manual, 1983, Section 3X) and (Arnold, not dated).

[8]"APL n. [...A Programming Language.] A language, devised by K. Iverson (1961), so compacted that the source code can be freely disseminated without revealing the programmer's intentions or jeopardizing proprietary rights."(Kelly-Bootle, 1981, p. 13)

3. Program L5.1

Program L5.1 implements the L5[9] language for mask level integrated circuit layout. It contains the defstruct definitions for the item and symbol data structures. CIF file output of these structures is performed by the function cifout. The functions in L5.1 are used by all layout functions in MSC. CIF output compatible with the Magic layout editor, SCMOS layout primitives, and a layout-label option in the item data structure are additions to MSC that support development of the microprogrammed controller described in Chapter VI.

4. Program defstructs.1

The defstructs.1 program contains definitions of object, extraction, and other internal data structures. Functions to support these data structures are all generated by the defstruct macro in lincoln.1.

5. Program front-page.1

The front-page program is loaded by the include function during compilation of MSC Lisp programs. This program establishes global variables required by MSC and uses the fasl function to load the lincoln.o, L5.o, and defstructs.o programs.

[9]For a description of the L5 layout language, see L5 User's Guide(MIT RVLSI-5, 7 March 1984).

6. Program prepass.l

The prepass.l program includes the top level functions of MSC. All MSC operations are controlled by the macpitts-compiler function in this program. Included in prepass.l are functions that process command line options. If the command line includes the "int" option, the macpitts-compiler function calls the interpret function in interpret.l to perform a functional simulation. If the command line includes the "cif" option, the macpitts-compiler uses other functions in prepass.l to read a design specification and library, extract definitions, evaluate evals, expand macros, and perform constant substitutions. The macpitts-compiler function uses functions in extract.l to convert the definitions into an extraction data structure. The macpitts-compiler function converts the extraction data structure into an object data structure to pass to the layout-object function in frame.l.

7. Program library

The library program is a text file that contains definition and eval data structures. The library program starts with a library definition followed by constant definitions, macro definitions, function definitions, test definitions, and organelle[10] definitions. The library definition marks the beginning of the library definitions in an

---

[10]"An organelle is the actual layout of a single bit of a function."(MIT RVLSI-3, 10 February 1983, p 25)

object. This mark is used to remove the library definitions from an object before it is written to an .obj file. The constant definitions and macro definitions are used to expand elements of the design specification file into components. Components are used to construct extraction data structures. The function definitions and test definitions are used by the interpret.l program to perform a functional simulation of the design specification and by the expand-form function in the prepass.l program to validate functions and tests in a design specification. The organelle definitions are used by the data-path.l program to instantiate an organelle (layout an instance of an organelle) and to provide information about the dimensions, connections, electrical characteristics, and logical function of an organelle. The library program also contains an eval function that loads the organelles.o program. The library program is interpreted at run time. Three UNIX directories are searched to find the library program. First, the current directory is searched for the library program. Next, a subdirectory of the user's home directory named macpitts is searched. Finally, the directory contained in the macpitts-directory Lisp variable is searched.[11] The first library program found by this search is read by the get-library function in the prepass.l program.

[11]For a description of the UNIX directory system, see (McGilton and Morgan, 1983, pp. 33-83).

8. <u>Program organelles.l</u>

The organelles.l program contains all of the technology dependent functions required to layout organelles in the data path. The nMOS version of organelles.l contains building blocks to construct basic inverter, NAND, and NOR gates. These gates are combined to form more complex functions. Input and output structures are added to the functions to form organelles. The SCMOS version of organelles.l does not use a hierarchical method to construct organelles. The SCMOS organelles are constructed as single items based on Magic[12] layouts. The functions in organelles.l are called by the organelle definitions in the library program. The compiled organelles.o program is loaded at run time by an eval in the library program.

9. <u>Program general.l</u>

The general.l program contains general purpose layout aids, a river router, controller track allocation functions, look up and query functions for data structures, and display functions. The layout aids are functions that produce nMOS superbuffers. The river router is a simple router used to construct the connections between the control logic array and the data path. The river router is also used by the layout-organelle function in the data-path.l program to construct the data path internal bus. The track

[12]Magic is a VLSI layout editor program.

allocation functions are used to determine the tracks (horizontal signal connections) that correspond to the required signals in the control logic array. The look up and query functions for data structures are used to access the definitions part of the object data structure. The display functions are used to display notes, warnings, statistics, and heralds. Note and warning functions print error and warning messages on the console. Statistic and herald functions print general information messages on the console. Printing of these messages is controlled by the stat, nostat, herald, and noherald command line options. Herald messages include the total Central Processor Unit (CPU) time and the amount of CPU time used for garbage collection. These times are displayed in sixtieths of a second.

10. Program extract.l

Program extract.l is used by the get-objectl function in the prepass.l program to decompose always and process components into lists of operand data structures. The operand data structures are converted into an extraction data structure. The always and process components are part of a design specification. The operand and extraction data structures are defined in the defstructs.l program. The four top level functions in the extract.l program that are called by the get-objectl function in prepass.l are get-sequencers-from-component-list, get-sequencers-required-definitions, extract-component-list, and post-process.

a. Get-sequencers-from-component-list

Get-sequencers-from-component-list is a recursive function that translates a list of components into a list of sequencer data structures. The sequencer data structure is defined in the defstructs.l program. A sequencer is a group of units in the data path that stores and manipulates the state number of a process[13]. The three types of sequencers are counter-stack, counter-no-stack, and no-counter-no-stack. These types of sequencers and the data path components that support them are listed in Table 3.2.

TABLE 3.2 SEQUENCER TYPES

	Counter-Stack	Counter-No-Stack	No-Counter-No-Stack
internal port	yes	yes	yes
state register	yes	yes	yes
incrementer	yes	yes	no
LIFO stack	yes	no	no

(1) No-Counter-No-Stack. The no-counter-no-stack sequencer is the simplest sequencer. The layout of the sequencer, produced by the layout-data-path defsymbol in the data-path.l program, includes a register unit, an

[13]For a discussion of processes and state numbers, see Paragraph II.D.

internal port unit, and a bit unit[14]. This type of sequencer is used for processes that use explicit go statements for all state transitions. The following process definition from a design specification has two states and uses explicit go statements. It requires a no-counter-no-stack sequencer.

```
(process ncns 0
    state1
          (par (setq a b) (go state2))
    state2
          (par (setq b c) (go state1)))
```

(2) Counter-No-Stack. The counter-no-stack sequencer is used for processes that do not use call and return statements and do not have explicit go statements for all state transitions. The layout of the sequencer, produced by the layout-data-path defsymbol in the data-path.l program, includes an incrementer type (1+) organelle unit in addition to the units used in a no-counter-no-stack sequencer. The following process definition from a design specification has two states and does not use call, return, or explicit go statements.

```
(process cns 0
    (setq a b)
    (setq b c))
```

(3) Counter-Stack. The counter-stack sequencer is used for processes that contain call and return

[14]For a description of the operation of the units in a sequencer, see Paragraph V.B.1.a.

statements. The layout of the sequencer, produced by the layout-data-path defsymbol in the data-path.l program, includes additional register units in addition to the units used in a no-counter-no-stack sequencer. The following process definition from a design specification has five states and uses call and return statements. The third element of the process definition specifies a stack depth of one so a single additional register unit is added by layout-data-path.

```
(process cs 1
   main
        (setq a b)
        (call sub)
        (go main)
   sub
        (setq b c)
        (return))
```

b. Get-sequencers-required-definitions

The get-sequencers-required-definitions function returns a list of register, internal port and signal, source, and destination definitions for the sequencers produced by the get-sequencers-from-component-list function. Tables 3.3, 3.4, and 3.5 list the definitions generated for each type of sequencer. At execution time, procname in these tables is replaced by the process name assigned in a design specification. A counter-stack sequencer requires definitions for a state register, internal go, call, and return signals, and a next-state internal port. In addition, a register definition is required for each level of

TABLE 3.3 COUNTER-STACK SEQUENCER DEFINITIONS

Definition Type	Definition Name
register	sequencer-procname-state
source	sequencer-procname-state
destination	sequencer-procname-state
internal signal	sequencer-procname-go
source	sequencer-procname-go
destination	sequencer-procname-go
internal signal	sequencer-procname-call
source	sequencer-procname-call
destination	sequencer-procname-call
internal signal	sequencer-procname-return
source	sequencer-procname-return
destination	sequencer-procname-return
internal port	sequencer-procname-next-state
source	sequencer-procname-next-state
destination	sequencer-procname-next-state
register	sequencer-procname-stack-#
source	sequencer-procname-stack-#
destination	sequencer-procname-stack-#

Note: The last three entries are repeated for each level of the LIFO stack with # replaced by the stack level.

TABLE 3.4 COUNTER-NO-STACK SEQUENCER DEFINITIONS

Definition Type	Definition Name
register	sequencer-procname-state
source	sequencer-procname-state
destination	sequencer-procname-state
internal signal	sequencer-procname-go
source	sequencer-procname-go
destination	sequencer-procname-go
internal port	sequencer-procname-next-state
source	sequencer-procname-next-state
destination	sequencer-procname-next-state

TABLE 3.5 NO-COUNTER-NO-STACK SEQUENCER DEFINITIONS

Definition Type	Definition Name
register	sequencer-procname-state
source	sequencer-procname-state
destination	sequencer-procname-state
internal port	sequencer-procname-next-state
source	sequencer-procname-next-state
destination	sequencer-procname-next-state

the LIFO stack. A counter-no-stack sequencer requires definitions for a state register, internal go signal, and a next-state internal port. A no-counter-no-stack sequencer requires definitions for a state register and a next-state internal port. All sequencers also require source and destination definitions for all registers, ports and signals. A source definition indicates that a data primitive is used to provide a value. A destination definition indicates that a data primitive is assigned a value.

c. Extract-component-list

The extract-component function converts a component-list into an extraction data structure. A component-list is a list of par, if, setq, nor, bit, go, call, and return components for each process or always in the design specification. A component-list also contains functions and tests from the design specification. The definitions of valid functions and tests are in the library program. The library program also contains macro and constant definitions that are used to convert a design specification into components. The extraction data structure is defined in the defstructs.l program.

d. Post-process

The post-process function expands and optimizes the extraction data structure. Each unit in the data path is assigned a unique unit number by the assign-bus-numbers function. Units are the word sized building blocks of the data path. The five types of units are register, output port, internal port, bit, and organelle. Units include multiplexers that are connected to the control logic array.

The multiplexers determine the input sources of the units. If the opt-d (data path optimization) command line option is selected, the order function in the order.l program is used to optimize the order of the units in the data path to minimize the number of tracks in the data path internal bus. The columns in the control logic array that represent control lines, test lines, drive lines, bit lines, and arg lines for the units in the data path are arranged to match the order of the units in the data path. These lines are the connections between the data path and the control logic array. This completes the construction of an extraction data structure.

11. Program order.l

Program order.l contains the top level order function that is used to optimize the placement of units in the data path and gates in the control logic array. The order function is called by the post-process function in extract.l to optimize the placement of units in the data path to reduce the number of tracks in the data path internal bus. The order function is called by the layout-object function in frame.l to optimize gate placement to reduce the number of tracks in the control logic array. Command line options beginning with "opt-" control the operation of the order function. The opt-d command line option enables optimization of unit placement in the data path. The opt-c command line option enables optimization of gate placement in the control logic array. The opt-n command line option disables the restriction that the order of columns in the control logic array must match the order of control lines, test lines, drive lines, bit lines, and arg lines for the units in the data path. The opt-n command line option should not be used with the current method of connecting the data path to the control logic array. The opt-n command line option may be used in the future if a method of connecting the control logic array to the data path that allows crossing of runs is added to MSC. The opt-p command line option enables printing of the current permutation and optimization status messages on the terminal. The opt-s command line option enables printing of a list of segments after the current permutation of the opt-p command line option. The opt-s command line option is inactive if the opt-p command line option is not specified. There are debugging functions named opt, order-this, this-node-column, and this-segment-nodes that demonstrate the operation of the order function.

12. Program frame.l

The frame.l program contains the top level function layout-object that is called by the macpitts-compiler function in prepass.l. The layout-object function converts an object data structure into an L5 item data structure. The item is converted into a CIF file by the cifout function in L5.l. The top level of the hierarchical item contains connections between the pads, data path, and control logic array, a set of L5 symbols that contain the lower level structures of the layout, and marks associated with the names of the pad signals. The lower level symbols are named layout-data-path, layout-control, layout-flags, layout-wing, layout-skeleton, and layout-pins. All symbols are created using the defsymbol macro in L5.l. The defsymbol definitions for layout-wing, layout-skeleton, and layout-pins are in frame.l. The defsymbol definition for layout-data-path is in data-path.l, the defsymbol definition for layout-control is in control.l, and the defsymbol definition for layout-flags is in flags.l.

13. Program data-path.l

The data-path.l program contains the L5 defsymbol macro definition for the layout-data-path symbol. The layout-data-path defsymbol is called by the layout-object function in frame.l. The top level of the L5 item data structure returned by layout-data-path contains all of the data path internal buses and a layout-unit symbol for each unit in the data path. Units are the word sized building blocks of the data path. The five types of units are register, output port, internal port, bit, and organelle[15]. Units include multiplexers that are connected to the control logic array. The multiplexers determine the input sources

[15]For a description of units, see Paragraph II.A.1.

of the units. The layout for all units in the data path is performed by the layout-unit function in data-path.l. The layout-unit function calls the layout-organelle function in data-path.l to build each bit of the unit. The layout-organelle function calls the lookup-gen-form function in general.l to find the layout of an organelle in the unit. The layout-organelle function then calls the layout-mpx function to layout the multiplexer, the layout-gen-form function to instantiate (generate an instance of) the layout of the organelle, and the river function in general.l to layout the data path internal bus. The functions that are called by the layout-mpx function generate four types of multiplexers: layout-mpx0, layout-mpx1, layout-mpx2, and layout-mpx3. The layout-mpx0 multiplexer has no constant (wired) inputs, the layout-mpx1 multiplexer has logical one constant (wired) inputs, the layout-mpx2 multiplexer has logical zero constant (wired) inputs, and the layout-mpx3 multiplexer has both logical zero and logical one constant (wired) inputs. Degenerate multiplexers with single inputs are produced by the layout-singleton-operand-list and the layout-mpx functions. The functions and defsymbols used to construct register, output port, internal port, and bit organelles and multiplexers are in data-path.l. The output port and internal port use the same layout-port-output function.

14. Program control.l

The control.l program contains the top level layout-control defsymbol definition that is called by the layout-object function in the frame.l program. The layout-control defsymbol returns a symbol containing the layout of the control logic array. The layout-control defsymbol merges the items returned by the layout-weinberger-gates function and the layout-weinberger-straps defsymbol to build the control logic array.

15. Program flags.l

The flags.l program contains the top level layout-flags defsymbol definition. The layout-flags defsymbol is called by the layout-object function in the frame.l program. Layout-flags returns a symbol item containing the registers for single bit data.

16. Program pads.l

The pads.l program is generated by the padgen.l program. Pads.l contains layout functions and defsymbols for all of the pad connections. The pads.l layout functions are called by the layout-pad function in frame.l. Since the pads.l program is machine generated, the file is deleted by the make clean Makefile command. The make pads.l, make pads.o, make macpitts, and make Macpitts Makefile commands cause the pads.l program to be generated.

a. Program padgen.l

The padgen.l program contains the top level main function that generates the pads.l program. The main function uses the read-cif function to read the pad20b and rinout pad CIF files.

b. Pad File pad20b

The pad20b pad CIF file contains CIF descriptions of the layout of four micron minimum feature size nMOS pads. This file is read by the read-cif function in the padgen.l program. The information in pad20b is used to generate the pads.l program.

c. Pad File rinout

The rinout pad CIF file contains CIF descriptions of the layout of five micron minimum feature size nMOS pads. This file is read by the read-cif function in the padgen.l program. The information in rinout is used to generate the pads.l program.

17. Program interpret.l

The interpret.l program contains the top level interpret function that is called by the macpitts-compiler function in prepass.l to perform a functional simulation of a design specification. The interpret function uses many of the same functions in the prepass.l and general.l programs that the macpitts-compiler function uses to get information from the design specification and library files. The function and test definitions in the library program are also used in the functional simulation of a design specification. The curses terminal screen updating and cursor movement optimization package functions in the c-routines.c program are used to display the results of the functional simulation on the terminal.

18. Makefile

The Makefile is used to generate the MSC compiler and maintain the files in the macpit directory. The Makefile uses the UNIX make facility to generate a program that is current with respect to the sources that are used to generate the program. If the program is current, the make commands do nothing. The Makefile contains commands to make the c-routines.o, lincoln.l, L5.o, defstructs.o, prepass.o, extract.o, frame.o, data-path.o, control.o, flags.o, padgen.o, pads.o, order.o, general.o, interpret.o, and organelles.o object files and the pads.l machine generated Lisp file. The make macpitts command generates an executable version of the MSC compiler with the name macpitts. The make Macpitts command installs the macpitts file with the name Macpitts on all three ISI workstations. The make Macpitts command also copies the current library and organelles.o files from the ISI0 workstation to the ISI1 and ISI2 workstations. The make xref command generates and prints a cross reference file named xref for the MSC compiler. The make clean command removes all excess files from the macpit directory on the ISI0 workstation. The make doc command prints all documentation and sources.

E. MSC INTERNAL STRUCTURE REQUIREMENTS

The MSC internal structure implies required characteristics of a controller. The controller must be constructed from hierarchical components. The combination of these components must be easily derived from an extraction or object data structure. The building blocks should be technology independent so that the same program can be used with different technologies. The software to produce the controller must be able to replace the corresponding programs and functions in the current nMOS MSC. These requirements will be used to evaluate the suitability of controller designs.

IV. CONTROLLER GOALS AND REQUIREMENTS

"Goals are targets for achievement, and serve to establish the framework for a software development project" and "Requirements specify capabilities that a system must provide in order to solve a problem."(Fairley, 1985, pp. 32, 33) It is crucial that goals and requirements be established in the analysis phase of a project before entering the design phase. In the design phase, goals are used to evaluate alternative design decisions and requirements are used to reject alternatives that will not solve the problem. In the implementation phase, the goals provide direction for construction of the structures specified in the design. In the testing phase, the requirements provide the acceptance criteria used to validate the performance of the system. The goals for this controller design are derived from the goals of the MSC development project and the requirements are based on the analysis performed in Chapters II and III.

A. MSC CONTROLLER GOALS

The primary goal of the MSC development project is to produce a version of MSC that is capable of producing an SCMOS VLSI circuit. An extension of this goal is to produce a silicon compiler that is capable of producing a VLSI circuit in any existing or future technology. MSC should be designed for easy addition of new technologies. As much of the system as possible should be technology independent. Technology dependent functions should be confined to the library and organelles.o programs that are loaded at run time. The SCMOS version of MSC should produce layouts that are compact and fast. Growth of the size of a layout should be a linear function of the complexity of the design specification for the layout. Geometric or exponential growth is to be avoided. Module interconnections should be simple. Modules should interconnect by abutting or overlapping whenever possible. The MOS transistors should be operated in the saturation or cutoff modes to reduce power consumption. Resistive pull up structures are to be avoided. Substrate and well contacts should be used in every cell to reduce the possibility of latch up. Long wiring runs, especially in polysilicon, are to be avoided. The purpose of these goals is to produce VLSI circuits that are small, fast, and efficient.

B. MSC CONTROLLER REQUIREMENTS

The MSC controller requirements are the results of the analysis in Chapters II and III of the target architecture and internal structure of MSC. To become part of a SCMOS silicon compiler, the controller must satisfy all of these requirements. The controller must be composed of multiple Mealy FSMs that have a single state number state variable and may include an incrementer and subroutine stack. Units in the data path are controlled by single FSMs using true multiplexers that have no side effects. All effects of multiplexer selection must be local to the output of the multiplexer and not global to the entire circuit. These FSMs must be constructed from hierarchical components and be easily derived from an extraction or object data structure. The software to produce the controller must be able to replace the corresponding programs and functions in the current nMOS MSC. These requirements will be used to eliminate design alternatives that do not satisfy the requirements and to validate the final controller implementation.

V. ARCHITECTURE TRADE-OFF STUDIES

The architecture trade-off studies are the start of the design phase of the controller development. In this first part of the design phase, different SCMOS logic structures and organizations[1] are evaluated to determine which are capable of fulfilling the controller requirements. After the structures and organizations that cannot meet the requirements are eliminated, those remaining are evaluated using the controller goals to determine which structure and organization have the greatest compliance with the goals. These are the structure and organization that will be used for the controller. The SCMOS logic structures that are evaluated are static CMOS complementary logic, pseudo-nMOS logic, dynamic CMOS logic, clocked CMOS logic, CMOS domino logic, cascade voltage switch logic, modified domino logic, and transmission gate logic. The SCMOS organizations that are evaluated are standard cells, programmable logic arrays (PLAs), Weinberger arrays, and microprogram read only memory (ROM).

[1]For a complete description of CMOS logic structures and organizations, see Chapters 5 and 8 in Principles of CMOS VLSI Design(Weste and Eshraghian, pp. 160-189 and 310-378, 1985).

A. LOGIC STRUCTURE TRADE-OFF STUDIES

The logic structure trade-off studies start with a requirements analysis to eliminate structures that do not meet the requirements. This is followed by a goals analysis to determine how closely each structure conforms to the controller goals. The structure with the closest conformance to the goals is the logic structure that will be used for the controller. All structures considered are homogeneous. Hybrid structures may be more efficient than the homogeneous structures but the programs that generate them would be more complex and circuits built of hybrid structures would not be regular. Hybrid structures would be a form of optimization that would sacrifice program maintainability for small gains in size or speed. Thus, hybrid structures will not be used in this controller.

1. Structural Requirements Analysis

The structural requirements analysis examines each structure and decides whether or not the structure is capable of satisfying the requirements. All structures that cannot satisfy the requirements are eliminated from further evaluation. The requirements used to screen structures are that the controller must be a modified Mealy FSM and that the controller must be constructed from hierarchical components. Basic logic functions of a structure must be easily cascaded or connected to form more complex logic functions in a hierarchical design.

a. Static CMOS Complementary Logic

Static CMOS complementary logic uses active pull up and pull down structures that are duals of each other. The pull up structure is composed of p-type transistors and the pull down structure is composed of n-type transistors. The duals of pull down structures connected in parallel are pull up structures connected in series and the duals of series pull down structures are parallel pull up structures. An example of a static CMOS complementary logic gate is in Figure 5.1. The output of this gate is the function (NOT ((A AND B) OR C)). The gate has twice as many transistors as there are inputs to the gate. The output is continuously responsive to changes in the inputs. This structure can be used to build a Mealy FSM. Basic logic functions of static CMOS complementary logic are easily cascaded or connected to form more complex logic functions in a hierarchical design. Static CMOS complementary logic is capable of satisfying all design requirements.

b. Pseudo-nMOS Logic

The pseudo-nMOS logic is similar to normal nMOS logic. The only difference is that the depletion mode n-type pull up transistor in nMOS is replaced by an enhancement mode p-type pull up transistor in pseudo-nMOS. An example of a pseudo-nMOS logic gate is in Figure 5.2. The output of this gate is the function (NOT ((A AND B) OR C)). The gate has one more transistor than there are inputs

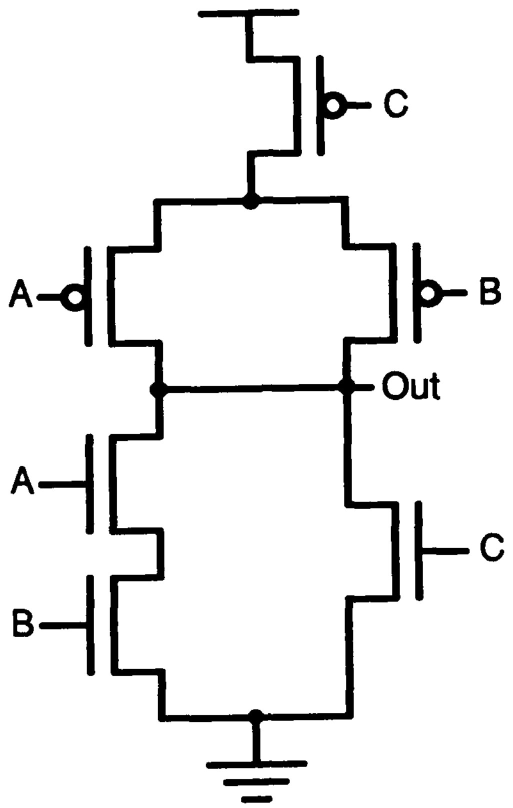


Figure 5.1 Static CMOS Complementary Logic

to the gate. This savings in the number of transistors is the result of eliminating the pull up structure that is the dual of the pull down structure in static CMOS complementary logic. The output is continuously responsive to changes in the inputs. The gate draws power whenever the pull down is active. This structure can be used to build a Mealy FSM. Basic logic functions of pseudo-nMOS logic are easily cascaded or connected to form more complex logic functions in a hierarchical design. Pseudo-nMOS logic is capable of satisfying all design requirements.

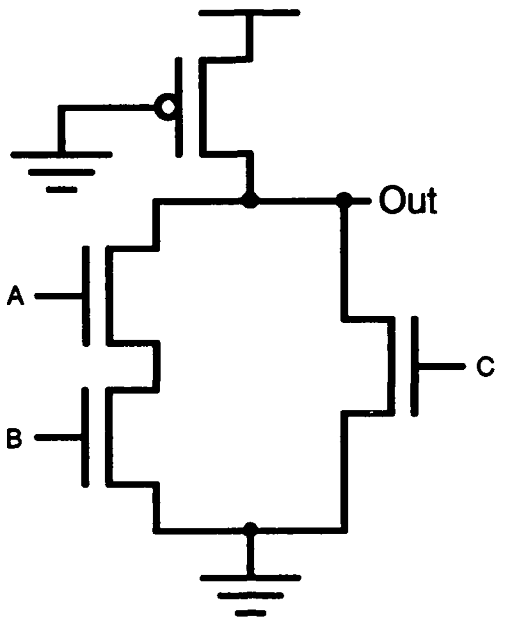


Figure 5.2 Pseudo-nMOS Logic

c. Dynamic CMOS Logic

Dynamic CMOS logic combines the simplicity of pseudo-nMOS with reduced power consumption. This structure takes advantage of the capacitance in the output node of the gate. Under control of a clock, the output node is alternately precharged to one and conditionally discharged to zero. If the logic function of the pull down structure is satisfied, the output node is pulled down to zero during the evaluation phase of the clock. If the logic function of the pull down structure is not satisfied, the output node remains a logic one during the evaluation phase of the clock. An example of a dynamic CMOS logic gate is in Figure 5.3. The output of this gate is the function (NOT ((A AND

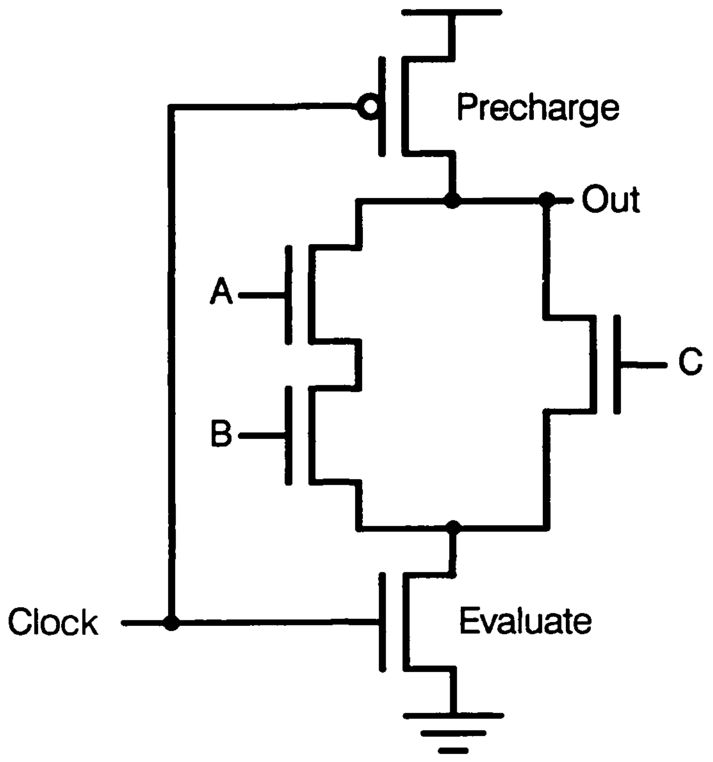


Figure 5.3 Dynamic CMOS Logic

B) OR C)). The gate has two more transistors than there are inputs to the gate. Dynamic CMOS gates are not easily cascaded. To reliably connect cascaded dynamic CMOS gate outputs to other dynamic CMOS gate inputs requires transmission gates and four phase clocking. One clock phase of delay from input to output is encountered for each level of logic. The output is not continuously responsive to changes in the inputs. During precharge, the output is independent of the input. During evaluation, the output may transition from one to zero if the pull down logic becomes satisfied. The output may not transition from zero to one during evaluation. This structure can be used to build a Moore FSM. This FSM would have a complicated clock cycle that may include several cycles of a four phase clock for every state transition. The dynamic CMOS logic could not be used to build a Mealy FSM. Basic logic functions of dynamic CMOS logic are not easily cascaded or connected to form more complex logic functions in a hierarchical design. Dynamic CMOS logic is not capable of satisfying all design requirements.

d. Clocked CMOS Logic

Clocked CMOS logic is static CMOS complementary logic with a clocked transmission gate added to the output. An example of a clocked CMOS logic gate is in Figure 5.4. The output of this gate is the function (NOT ((A AND B) OR C)). The gate has two more transistors than two times the number of inputs to the gate. Clocked CMOS logic is more complex and slower than static CMOS complementary logic. The output of the gate is not responsive to the input to the gate when the clock is zero. Clocked CMOS logic could be used to build a Moore FSM but not a Mealy FSM. Basic logic functions of clocked CMOS logic are easily cascaded or connected to form more complex logic functions in a hierarchical design. Clocked CMOS logic is not capable of satisfying all design requirements.

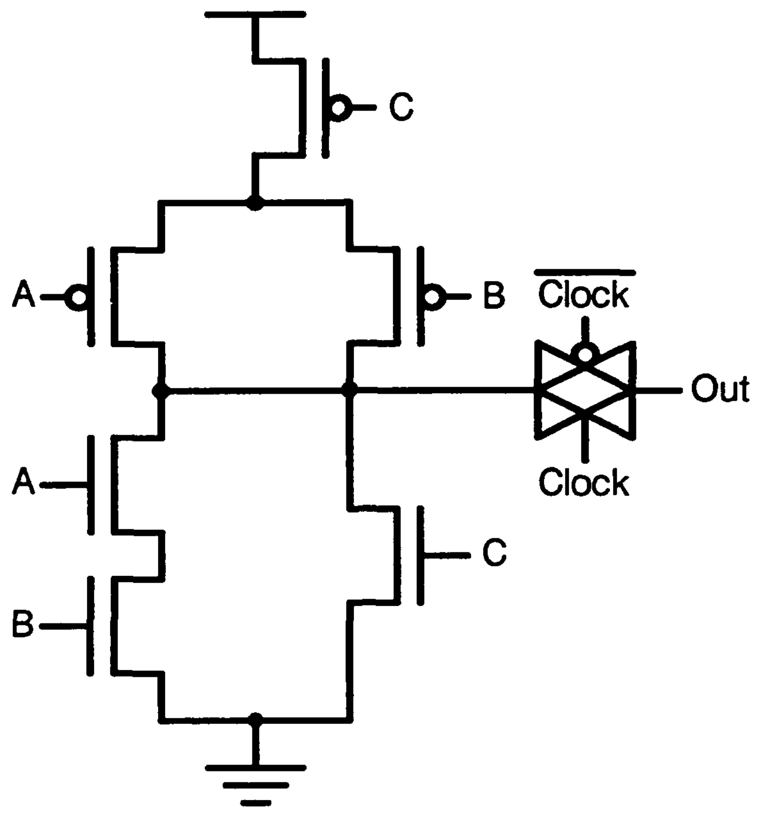


Figure 5.4 Clocked CMOS Logic

e. CMOS Domino Logic

CMOS domino logic gates are dynamic CMOS logic gates with static CMOS complementary logic inverters on the output. By inverting the output of each gate, CMOS domino logic gates can be easily connected in cascade configurations. An example of a CMOS domino logic gate is in Figure 5.5. The output of this gate is the function ((A AND B) OR C). The gate has four more transistors than there are inputs to the gate. The output is not continuously

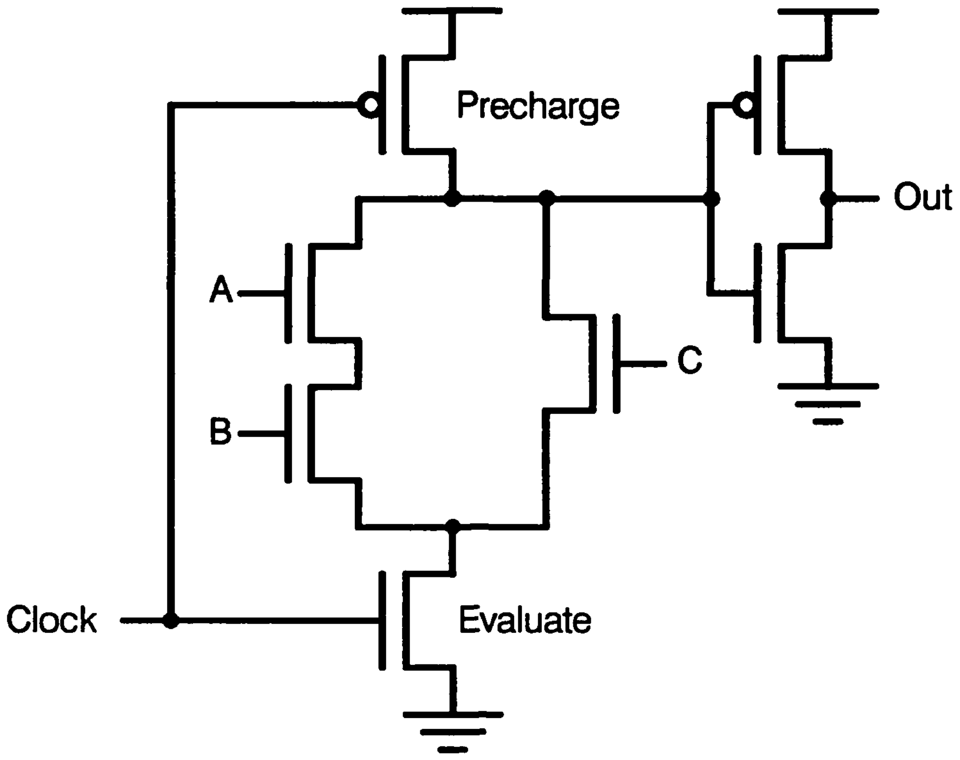


Figure 5.5 CMOS Domino Logic

responsive to changes in the inputs. During precharge, the output is independent of the input. During evaluation, the output may transition from zero to one if the pull down logic becomes satisfied. The output may not transition from one to zero during evaluation. This structure can be used to build a Moore FSM. The CMOS domino logic could not be used to build a Mealy FSM. Basic logic functions of CMOS domino logic are easily cascaded or connected to form more complex logic functions in a hierarchical design. CMOS domino logic is not capable of satisfying all design requirements.

f. Cascade Voltage Switch Logic

Cascade voltage switch logic requires complementary inputs and produces complementary outputs. Each gate uses two n-type pull down structures that are the duals of each other. The output of each pull down structure is connected to the gate of a single p-type pull up transistor for the other output. This form of cascade voltage switch logic is slower than static CMOS complementary logic. For a short time during switching, the pull up and pull down structures for one of the outputs are both active. An example of a cascade voltage switch logic gate is in Figure 5.6. The output of this gate is the function (NOT ((A AND B) OR C)). The gate has two more transistors than twice the number of inputs to the gate. The output does respond continuously to changes in the input signals. A more efficient version of cascade voltage switch logic is clocked with precharge and evaluate transistors similar to dynamic CMOS logic on each pull down structure and inverters on each output like CMOS domino logic. The outputs of the clocked cascade voltage switch logic gates are not continuously responsive to changes in the inputs. During precharge, the outputs are independent of the inputs. During evaluation, the outputs may transition from zero to one if their pull down logic becomes satisfied. The outputs may not transition from one to zero during evaluation. The static version of cascade voltage switch logic could be used to construct a Moore or

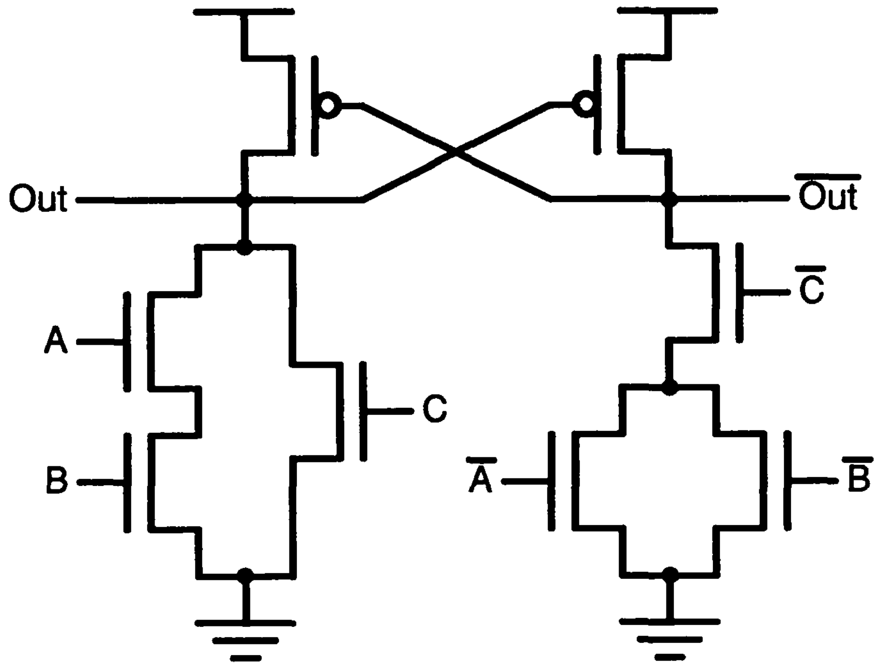


Figure 5.6 Cascade Voltage Switch Logic

Mealy FSM. The clocked version of cascade voltage switch logic could be used to construct a Moore FSM but not a Mealy FSM. Basic logic functions of cascade voltage switch logic are easily cascaded or connected to form more complex logic functions in a hierarchical design. The static version of cascade voltage switch logic can satisfy the controller requirements and the clocked version of cascade voltage switch logic cannot satisfy the controller requirements.

g. Modified Domino Logic

Modified domino logic is an improvement on CMOS domino logic. In modified domino logic, cascade combinations of gates are composed of alternate p-type and n-type gates. The n-type gates are similar to CMOS domino logic gates without the inverters on the outputs. The p-type gates are made of combinations of single n-type transistors to precharge the outputs to zero and p-transistor pull up structures that are the duals of the n-type structures for the same gate function. The clock for the p-type gates is the inverse of the clock for the n-type gates. By eliminating inverter outputs, the layout of modified domino logic is more compact than CMOS domino logic. The operation of the modified domino logic is similar to CMOS domino logic. Modified domino logic could be used to construct a Moore FSM but not a Mealy FSM. Basic logic functions of modified domino logic are easily cascaded or connected to form more complex logic functions in a hierarchical design. Modified domino logic cannot meet the controller requirements.

h. Transmission Gate Logic

Transmission gate logic uses combinations of transmission gates to build logical functions. A CMOS transmission gate is a parallel connection of a p-type transistor and an n-type transistor. The gate of the n-type transistor is driven by the control signal and the gate of the p-type transistor is driven by the complement of the control signal. The transmission gate is like an electronic relay that conducts when the control signal is a logic one and does not conduct when the control signal is a logic zero. Many logical functions may be efficiently implemented using transmission gate logic. An implementation of an exclusive or (XOR) gate requires only two transmission gates. Some logical functions are accomplished more efficiently using other structures like static CMOS complementary logic. An example of a transmission gate logic gate is in Figure 5.7. The output of this gate is the function (NOT ((A AND B) OR C)). This logic function is realized using four transmission gates. Another representation of the same gate using discrete transistors is in Figure 5.8. An n-type transistor transmits a strong zero and a

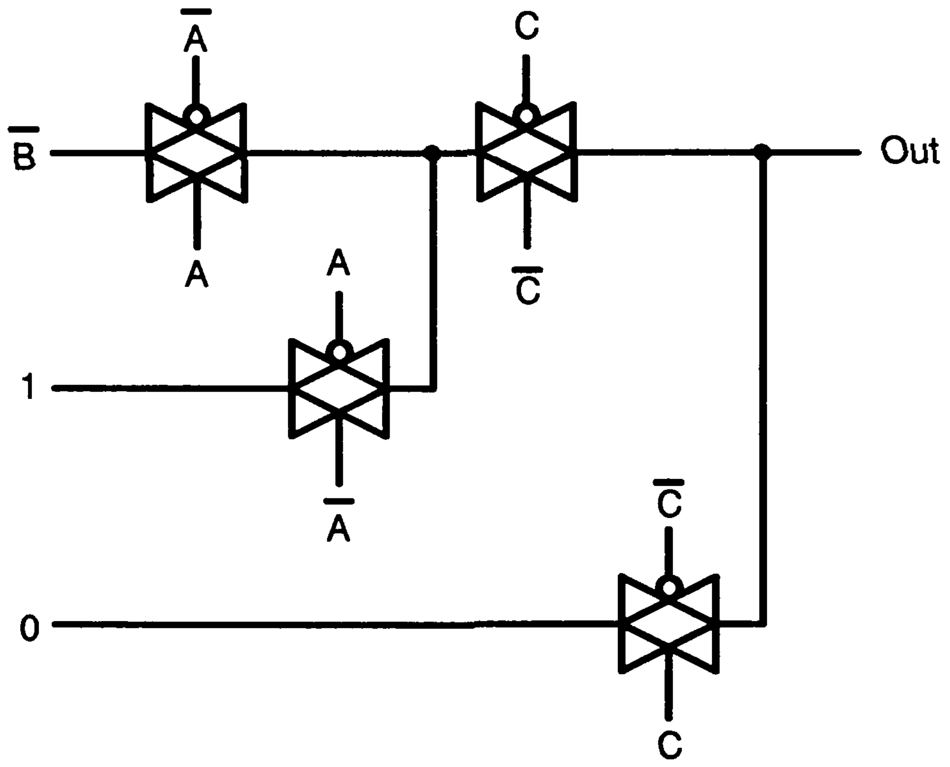


Figure 5.7 Transmission Gate Logic

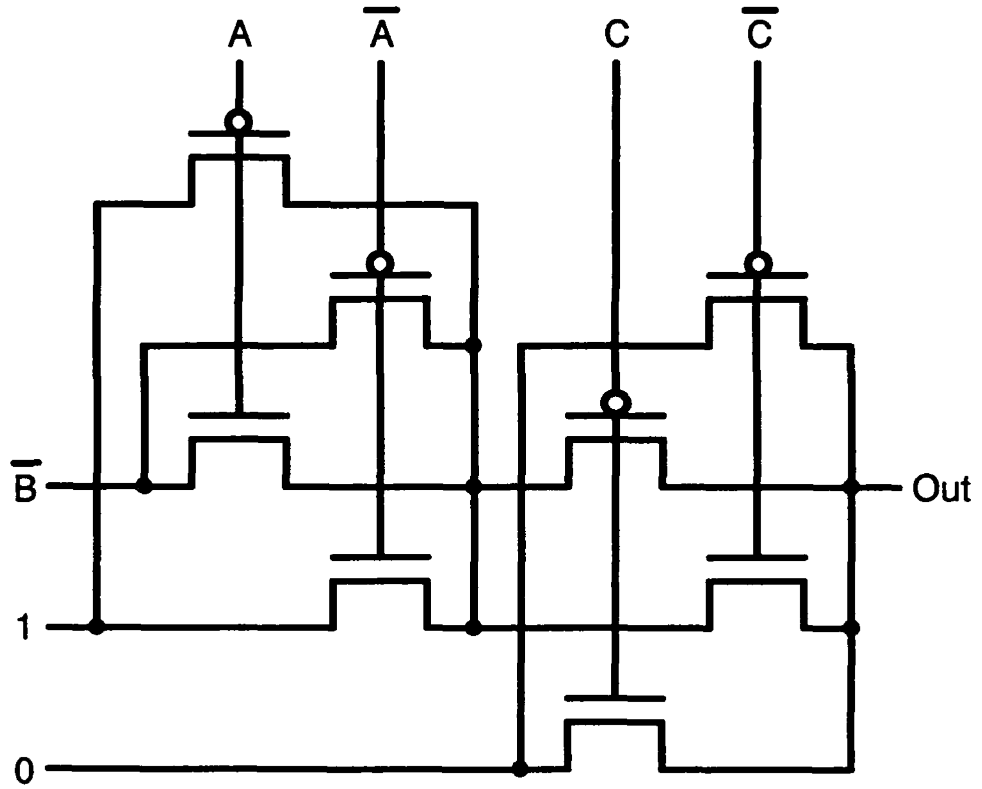


Figure 5.8 Transmission Gate Discrete Components

weak one and a p-type transistor transmits a strong one and a weak zero. Transmission gates for constant inputs require only the transistors that have strong transmission characteristics for the constant. Thus, the n-type transistors associated with constant one inputs and the p-type transistors associated with constant zero inputs are not required. The circuit in Figure 5.8 may be reduced to the circuit in Figure 5.9 by eliminating the transistors with weak transmission characteristics for constant inputs. For this sample circuit, the transmission gate logic gate uses the same number of transistors as the static CMOS complementary logic

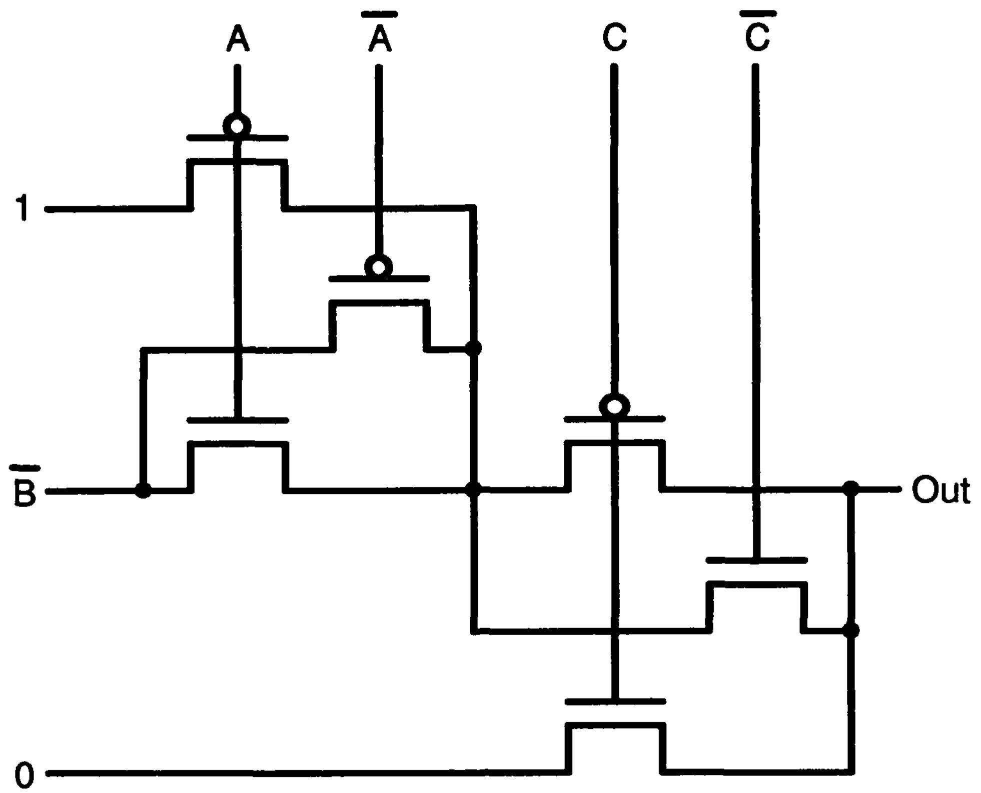


Figure 5.9 Transmission Gate Logic Reduced Circuit

circuit. The p-type transistors in the static CMOS version may be replaced by a single p-type precharge transistor for dynamic logic. A static version of the gate using a single p-type pull up transistor that is driven by an inverter connected to the output is another variant of this circuit. The size ratios of the transistors are critical in the version of the circuit with a pull up transistor and inverter. The static variations of transmission gate logic could be used to construct a Mealy FSM. The dynamic variation could not be used to construct a Mealy FSM. Basic logic functions of the full complementary form of transmission gate logic are easily cascaded or connected to form more complex logic functions in a hierarchical design while the other forms of transmission gate logic are not easily cascaded. The complementary transmission gate logic can satisfy controller requirements while the dynamic transmission gate logic cannot satisfy the requirements.

i. Acceptable Structures

The structures that are capable of satisfying the controller requirements are static CMOS complementary logic, pseudo-nMOS logic, the static version of cascade voltage switch logic, and the static versions of transmission gate logic. These are all structures that can be used to construct Mealy FSMs.

2. Structural Goals Analysis

The logic structures that remain after requirements analysis are next examined to determine how closely each structure conforms to the controller goals. The structure that is nearest to the controller goals is selected for implementation of the controller.

a. Static CMOS Complementary Logic

The goals that are closely satisfied by static CMOS complementary logic are high speed, low power, and simple interconnection. High speed is obtained by active pull up and pull down structures. Low power results from all transistors being operated in the saturation or cutoff modes. Simple interconnection is achieved since complementary signals are not required. The goal that is not closely satisfied by static CMOS complementary logic is size. The medium size of static CMOS complementary logic is the result of the dual pull up and pull down structures.

b. Pseudo-nMOS Logic

The goals that are closely satisfied by pseudo-nMOS logic are size and simple interconnection. The small size is the result of the single p-type transistor for the pull up of each gate. Simple interconnection is achieved since complementary signals are not required. The goals that are not satisfied by pseudo-nMOS logic are low power and high speed. The pull up structure that is active when the pull down structure is in saturation causes high power. For proper operation, the pull up transistors must be weak compared to the pull down transistors. This results in slow rise times for the outputs of gates. The slow rise times reduce the speed of the circuit.

c. Static Cascade Voltage Switch Logic

The only goal satisfied by static cascade voltage switch logic is low power. Even the low power is not as good as that obtained using static CMOS complementary logic and complementary transmission gate logic. The transistors are in the active mode during the contention period that is part of switching. The goals that are not satisfied by static cascade voltage switch logic are small size, high speed, and ease of interconnection. Cascade voltage switch logic circuits use more transistors than static CMOS complementary logic circuits. The speed of cascade voltage switch logic is reduced by the contention period that is part of switching. Cascade voltage switch logic circuits require complementary inputs that result in more complex interconnections.

d. Complementary Transmission Gate Logic

The goals that are closely satisfied by complementary transmission gate logic are small size, low power, and high speed. The size of complementary transmission gate logic circuits ranges from the same size as static CMOS complementary logic to smaller than pseudo-nMOS depending on the circuit logic function. The power consumption is almost as low as static CMOS complementary logic and the speed is almost as fast. The slightly slower speed and slightly higher power are the result of more capacitance in the increased number of connections between the pull up and pull down regions and the greater number of separate source and drain regions. In static CMOS complementary logic, transistors of the same type that are connected in series may have their common source and drain nodes merged together and transistors connected in parallel may merge their sources or drains. This significantly reduces the total node capacitance. It is very difficult to merge sources or drains in complementary transmission gate logic circuits.

The goal that is not supported by complementary transmission gate logic is simple interconnection. Complementary transmission gate logic circuits require complementary inputs that result in more complex interconnections.

3. Structure Selection

The structure that satisfies all requirements and comes closest to satisfying all goals is static CMOS complementary logic. This structure will be used to implement the controller.

B. LOGIC ORGANIZATION TRADE-OFF STUDIES

There are two organizational design decisions that yield four possible organizations for the controller. The first design decision is whether or not to use data path organelles with similar functions to the organelles used in the nMOS version of MSC. The second design decision is whether to use standard cells or a microprogram read only memory (ROM) for the MSC FSM combinational logic. Programmable logic arrays (PLAs) and Weinberger arrays are rejected since they cannot be easily implemented using the static CMOS complementary logic structure. These arrays generate complex logical functions by wired logic connections of single pull down transistors. This wired logic works with dynamic CMOS logic and pseudo-nMOS logic. Wired logic does not work with static CMOS complementary logic.

PLAs and Weinberger arrays are normally constructed using dynamic CMOS logic or pseudo-nMOS logic.

1. Data Path Units

The data path units used in the current nMOS MSC controller include register, organelle, bit, and internal port units[2]. As discussed in Paragraph III.D.10.a, each process in the design specification has a sequencer in the data path. The data path sequencer contains all of the elements of the MSC FSM shown in Figure 2.9 except the combinational logic. The combinational logic for the nMOS MSC FSM is in a Weinberger array.

a. Sequencer Types

There are three types of sequencers: counter-stack, counter-no-stack, and no-counter-no-stack.

(1) No-Counter-No-Stack. The no-counter-no-stack sequencer is the simplest sequencer. This type of sequencer is used for processes that use explicit go statements for all state transitions. Figure 5.10 is a block diagram of a no-counter-no-stack sequencer for the code fragment in Section III.D.10.a(1). It is constructed from a register unit, an internal port unit, and a bit unit[3]. There is a control line from the combinational logic to the

[2]The organization and function of units is discussed in Paragraph II.A.1 and the functions used to layout the units and data path are discussed in Paragraph III.D.13.

[3]For a discussion of the internal structure of units, see Section II.A.1.

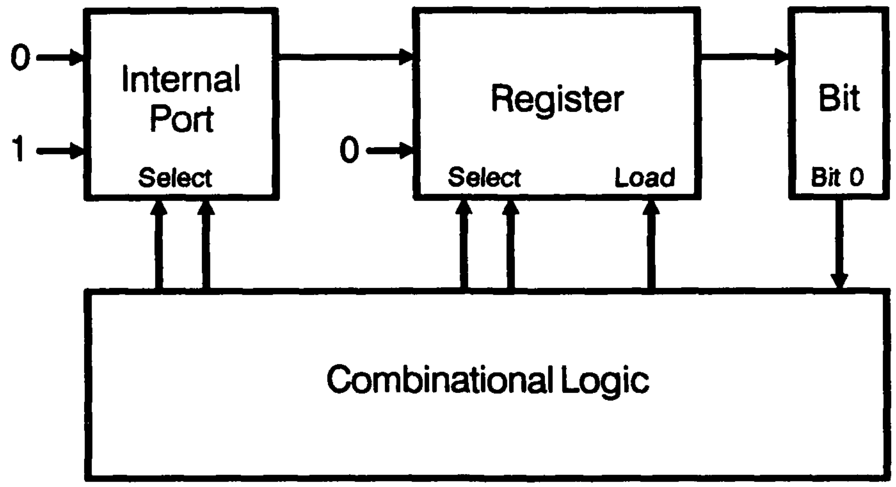


Figure 5.10 No-Counter-No-Stack Sequencer

sequencer for every input to every unit to select multiplexer inputs and an additional control line to the register unit to enable loading the register. The register unit contains a register that stores the current state number and a multiplexer that selects the source of the next state number. The constant (hardwired) zero input to the register unit is selected when the reset signal is high. The internal port unit is used to generate the go state numbers that are stored in state number memory. The go state number inputs to the internal port unit are hardwired constant values. Each state that can be the result of an explicit go state transition has a constant input to the internal port unit. Using the internal port, each single bit control line from the control logic array selects one integer state number that has as many bits as the data path word size. The bit unit is used to extract groups of bits from the present state number that is stored in the register unit. This extraction converts word sized data from the data path into bit sized data for the combinational logic. The bits are used in the combinational logic to determine what control signals to apply to the register and internal port units. Only the bits that are required to uniquely represent all state numbers in the process are extracted. For example, if the data path word size is eight bits and a process only has two states, the least significant bit (LSB) of the output of the register unit will be extracted by the bit unit. This one bit can uniquely identify all of the valid state numbers. The seven most significant bits of the register and internal port units are not used by the controller.

(2) Counter-No-Stack. The counter-no-stack sequencer is used for processes that do not use explicit go statements for all state transitions. Figure 5.11 is a block diagram of a counter-no-stack sequencer for the code fragment in Section III.D.10.a(2). The counter-no-stack sequencer contains an organelle unit and all of the units in the no-counter-no-stack sequencer. The organelle unit performs the increment (1+) function. The input to the organelle unit is connected to the register unit output and the output of the organelle unit is connected to one of the inputs to the register unit. An additional internal signal

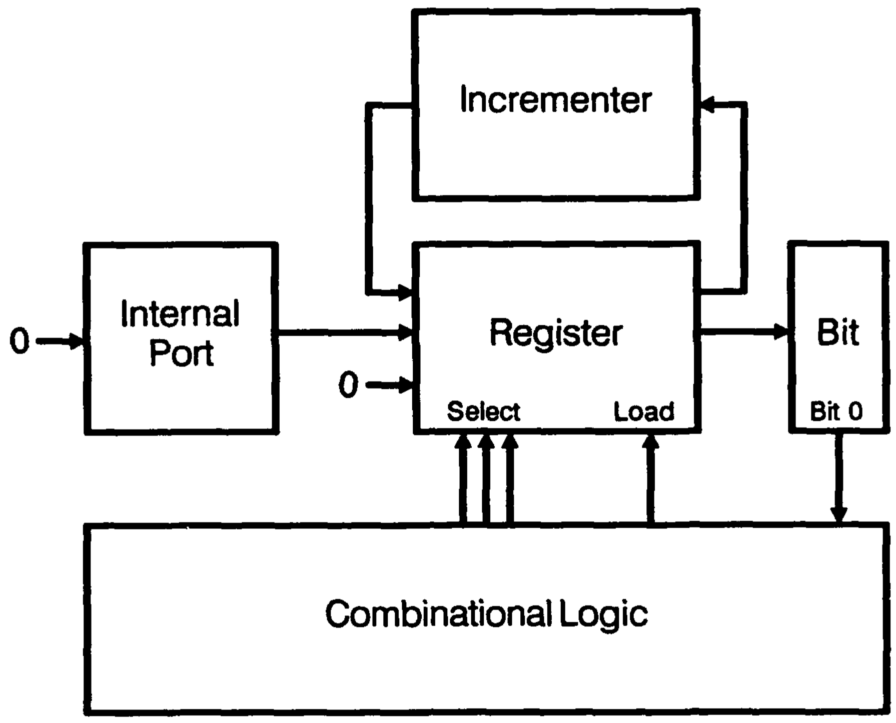


Figure 5.11 Counter-No-Stack Sequencer

named sequencer-procname-go[4] is used inside the combinational logic to determine whether to select the internal port unit or the organelle unit input to the register unit. The input selection is controlled by the multiplexer inside the register unit.

(3) Counter-Stack. The counter-stack sequencer is used for processes that contain call and return statements. Figure 5.12 is a block diagram of a counter-stack sequencer for the code fragment in Section

[4]Procname in this signal name is replaced at execution time by the process name assigned in the design specification.

III.D.10.a(3). The counter-stack sequencer contains additional register units and all of the units in the counter-no-stack sequencer. The additional register units form a last-in-first-out (LIFO) stack. Since the code fragment in Section III.D.10.a(3) specifies a stack depth of one, a single register unit is added for the LIFO stack in Figure 5.12. The output of the organelle unit is pushed onto the stack and the stack is popped to generate one of the inputs to the register unit that contains the state number memory. Internal signals named sequencer-procname-call and

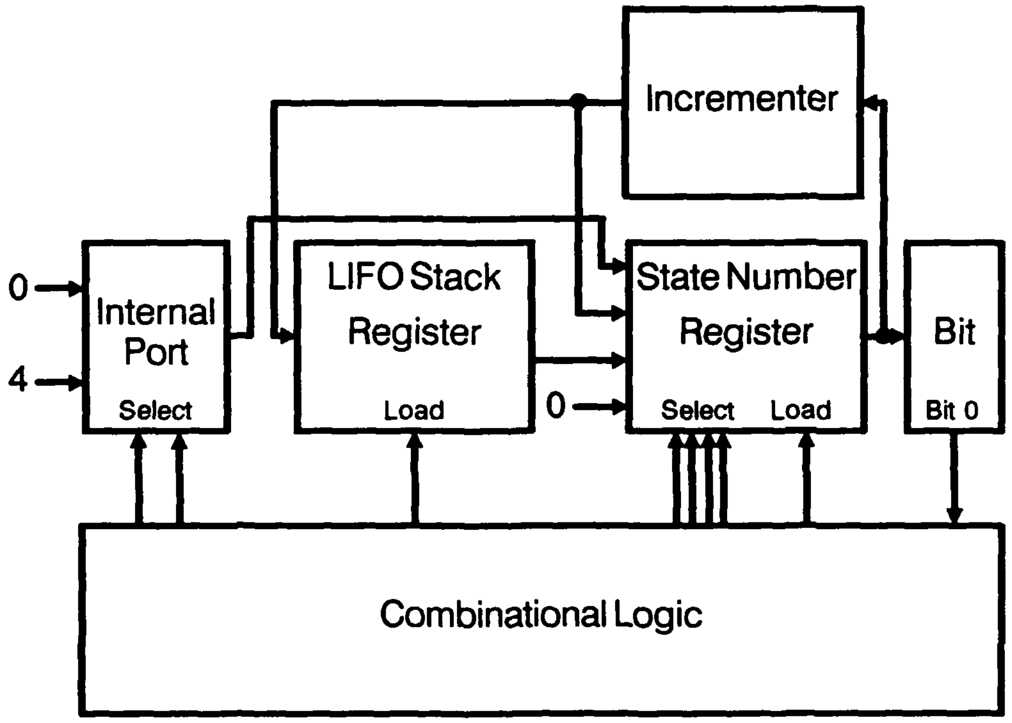


Figure 5.12 Counter-Stack Sequencer

sequencer-procname-return[5] are used inside the combinational logic to select an input to the state number memory register unit and to control pushing and popping the LIFO stack formed by the additional register units.

b. Data Path Unit Advantages.

The data path units provide a useful abstraction of the functions that are required to manipulate the state number of an MSC FSM. Data path units use existing data path software to generate a large part of the controller. All units can be located in a single data path using the current MSC floor plan or a second controller data path could be generated using the existing layout-data-path defsymbol in the data-path.l program[6]. Generation of a separate controller data path requires modification of the .obj file structure. The sequencer units would be removed from the data-path section of the .obj file and placed in a new control-data-path section. This would require modification of the defstructs.l, prepass.l, and extract.l programs. The existing layout-data-path defsymbol could be used to layout the data-path and control-data-path symbols. A separate controller data path does not have to use the same data path word size as the arithmetic and logical data path. Having a

[5]Procname in these signal names is replaced at execution time by the process name assigned in the design specification.

[6]For a description of the layout-data-path defsymbol in the data-path.l program see Paragraph III.D.13.

separate controller data path saves space if the number of bits required to represent the largest state number is less than the data path word size. With a single data path, the maximum number of states in a process is limited by the largest number of integers that can be represented with the data path word size. A separate controller data path allows a process to have more states than the largest number of integers that can be represented with the data path word size.

c. Data Path Unit Alternatives

If data path units are not used, each of these sequencer functions must be constructed from standard cells or added to the combinational logic. Construction of these functions from standard cells requires a large number of new and modified functions in the MSC programs. Adding these functions to the combinational logic would increase the complexity of the functions used for generation of the combinational logic. The alternative with the least risk and minimum modification of existing software is to use data path organelles.

2. Standard Cell or Microprogram ROM

Either standard cells or a microprogram ROM may be used to implement the combinational logic of the MSC FSM.

a. Standard Cell

Use of standard cells requires that all multiplexer and register control signals be expressed as functions of the values of signals, ports, registers, and flags. Generation of these expressions requires significant processing of the information in a design specification. Efficient placement and routing for standard cells that implement the control signal functions is a very difficult programming problem. The size of the combinational logic, a function of the number of standard cells and amount of interconnection, is not a linear function of the design specification complexity. Geometrical or exponential growth in size is expected for more complex design specifications.

b. Microprogram ROM

A microprogram ROM has many characteristics that make it an attractive choice for the combinational logic of the MSC FSM. The structure is simple and regular. It can be constructed entirely of modules that connect by abutting or overlap. This significantly reduces routing problems. The required contents of the ROM are easily derived from the design specification. There is a one-to-one correspondence of the states and sub-states in a design specification to the rows in the ROM. Design specification sub-states are the alternative outputs of the combinational logic section that are the results of selecting different alternatives for cond statements in the design specification. The vertical dimension of the ROM is a linear function of the number of states and sub-states in a design specification. There is a row in the ROM for each state and sub-state. The horizontal dimension of the ROM is a linear function of the number of control signals required by the data path, number of cond statements in the design specification, and the number of bits required to represent the largest state number. There is a column in the ROM for each control signal required and each cond statement. There is a column of row decoder cells for each bit required to represent the largest state number. The number of logic delays in the row decoders is a function of the number of bits required to represent the maximum state number. This is a logarithmic function of the number of states in a process.

C. LOGIC STRUCTURE AND ORGANIZATION SELECTION

The selections for logic structure and organization of the controller are static CMOS complementary logic for the logic structure and microprogram ROM for the controller organization. The speed, size, low power, and simple interconnection of static CMOS complementary logic are the features leading to its selection. The simple regular structure, simplified interconnection, high level of correspondence to the design specification, and orderly growth of the microprogram ROM are the features leading to its selection. A summary of the evaluations of each of the logic structure and organization options for the controller is listed on the next page.

Logic Structures

Static CMOS Complementary Logic

* Can satisfy all controller requirements
* High speed
* Medium size
* Small static power consumption
* Simple interconnection

Pseudo-nMOS Logic

* Can satisfy all controller requirements
* Medium speed
* Small size
* Large static power consumption
* Simple interconnection

Dynamic CMOS Logic

* Cannot use in Mealy FSM

Clocked CMOS Logic

* Cannot use in Mealy FSM

CMOS Domino Logic

* Cannot use in Mealy FSM

Cascade Voltage Switch Logic

* Static form can satisfy all controller requirements
* Medium speed
* Large size
* Low power
* Complex interconnections

Modified Domino Logic

* Cannot use in Mealy FSM

Transmission Gate Logic

* Static form can satisfy all controller requirements
* High speed
* Small size
* Low power
* Complex interconnections

Organizations

Data Path Units

* Useful functional abstraction
* Utilize existing software

Standard Cell Combinational Logic

* Complex program for placement and routing
* Geometrical or exponential growth

Microprogram ROM Combinational Logic

* Simple regular structure
* Simple program for ROM construction
* Linear or logarithmic growth

VI. CONTROLLER DESIGN

The technology independent part of the design process for the MSC microprogram ROM controller involves partitioning controller functional requirements into groups, allocating groups of requirements to units, establishing floor plans to describe the physical relationship of the units, dividing the units into cells, and specifying the functions performed by each cell. No internal circuitry is specified in the technology independent design. The technology dependent part of the design process involves specifying the internal signal conventions and circuitry of each cell in the design.

A. FUNCTIONAL REQUIREMENTS PARTITIONS

The MSC microprogram ROM controller must include all functions required to implement MSC finite state machines whose structure is shown in Figure 2.9. The controller must be able to store a current state number. This storage must be clocked so that the current state number changes during state transition and remains constant at all other times. There must be a facility for selection of the source of the next state number to be loaded into storage. There must be a capability to generate next state numbers for go and call statements in the design specification. There must be a FILO stack for storage of return state numbers. There must be an incrementer to generate a next sequential state number. There must be a combinational logic section that generates control signals for the data path and the other sections of the controller. The outputs of this combinational logic section must be functions of the current state number and signals from the data path. All requirements that are satisfied by the control logic array in the nMOS version of MSC form one group of requirements and the requirements satisfied by data path units in the nMOS version of MSC form the second group. The first group contains the combinational logic requirements and the second group contains all remaining requirements.

B. UNIT IDENTIFICATION

The partitioned controller functional requirements are allocated to units that may be located in the data path or microprogram ROM.

1. Data Path Units

There are five controller units located in the data path: the process state number register, the next state number internal port, the bit, the FILO stack registers and the increment organelle units. These units form sequencers as described in Sections III.D.10.a and V.B.1. All of these units are located in the single data path of the current nMOS MSC. In future versions of MSC, they may be located in the same data path or in a separate controller data path.

2. <u>Microprogram ROM Units</u>

The microprogram ROM must produce outputs that are functions of test signals from the data path or pads and the current state number stored in the process state number register data path unit. Three units support this processing: the ROM array unit, the row decoder unit, and the conditional unit. Additional buffer units are added for efficiency. The configuration of these units is shown in Figure 6.1. All these units are located in the microprogram ROM section of the layout.

a. ROM Array

The ROM array unit contains the microprogram code for the controller. The array is organized in horizontal rows of words and vertical columns of bits. Each row of the ROM array is one word of memory that is associated with one state or substate of a process in the design specification. Each column of the ROM array generates a one bit signal that is used internally by the microprogram ROM, connected to data or control lines for pads, or connected to control lines for the data path.

b. Row Decoder

The row decoder unit selects a row in the ROM array that is associated with a state in a process defined in the design specification. One row is selected for each process based on the current state number stored in the process state number register associated with the process.

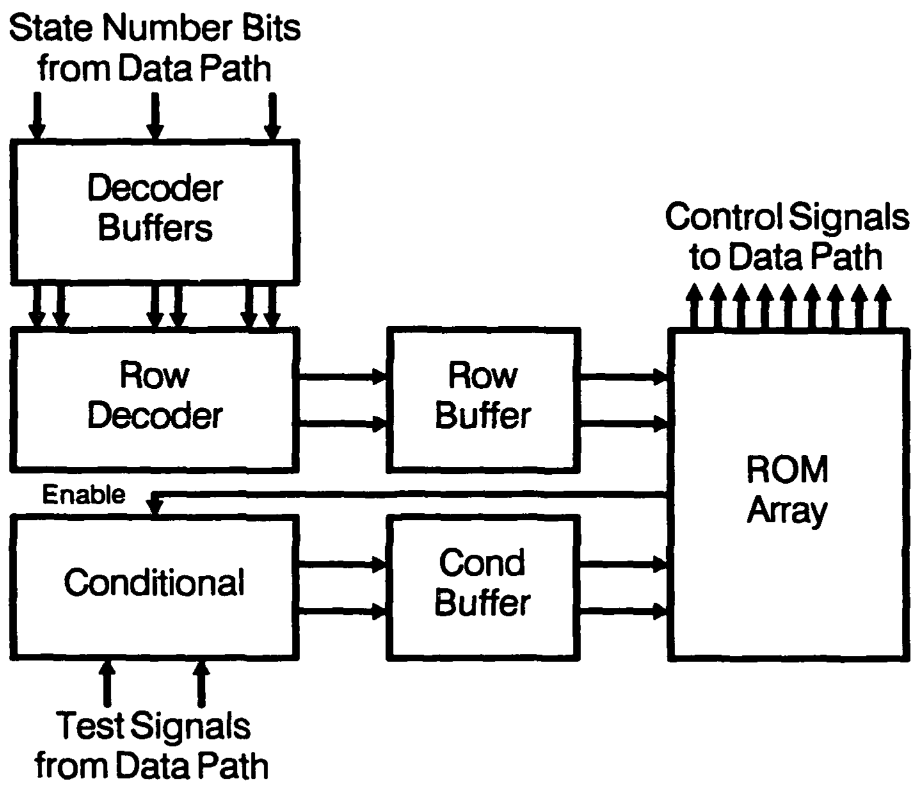

State Number Bits from Data Path
Decoder Buffers
Control Signals to Data Path
Row Decoder
Row Buffer
ROM Array
Enable
Conditional
Cond Buffer
Test Signals from Data Path

Figure 6.1 Microprogram ROM Units

There is a row decoder unit for each process in the design specification.

c. Conditional

The conditional unit selects a row in the ROM array based on signals from the data path or pads. There is a conditional unit for each cond statement in the design specification. The conditional unit has an input for each alternative in the cond statement in the design specification. The inputs to the conditional unit are connected to test lines from the data path or pads. The test lines may have logical one or zero signals based on conditions in the data path. No more than one row in the ROM array will be selected by each conditional unit. Each conditional unit is enabled by a unique one bit column of the ROM array. If the bit column for the conditional unit is a logic zero, the conditional unit will not select any row in the ROM array. If the bit column for the conditional unit is a logic one, then the first row in the ROM array that is associated with a logic one input to the conditional unit will be selected. If the last alternative in the cond statement in the design specification is a default true (t) and all of the inputs to the conditional unit are logic zero, the last row in the ROM array associated with the conditional unit will be selected.

d. Buffer

Buffer units are used on the inputs to the row decoder units, between the row decoder units and the ROM array, and between the conditional units and the ROM array. The buffers convert single input signals into complementary signals and may provide increased drive current.

C. FLOOR PLAN

Two floor plans are established for the microprogram ROM controller. The first floor plan is for the organization of the complete controller and the second is for the internal floor plan of the microprogram ROM.

1. Controller Floor Plan

There are several floor plans that are available for the microprogram ROM controller. The first floor plan, shown in Figure 6.2, is similar to the current nMOS MSC floor plan with the control logic array replaced by the microprogram ROM. The data path units for the sequencers of the controller are located in the common data path. The data path is located above the microprogram ROM. Alternative floor plans, shown in Figures 6.3 and 6.4, use a separate control data path. The separate control data path may be located to the left of the microprogram ROM or below the microprogram ROM. The arithmetic and logical data path is still located above the microprogram ROM. Any one of these floor plans is suitable for the microprogram ROM controller. The separate data paths can be generated by

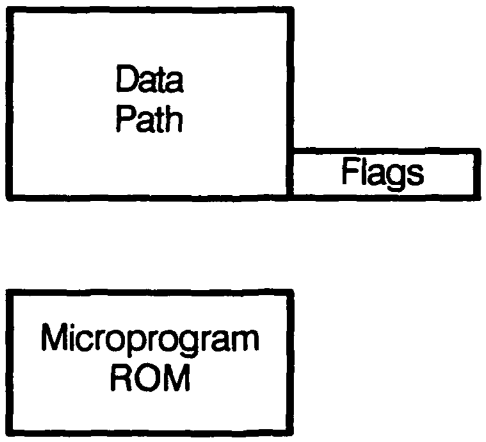


Figure 6.2 Controller Floor Plan

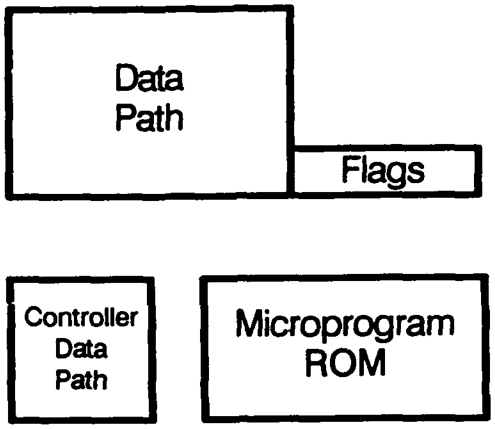


Figure 6.3 Alternate Controller Floor Plan

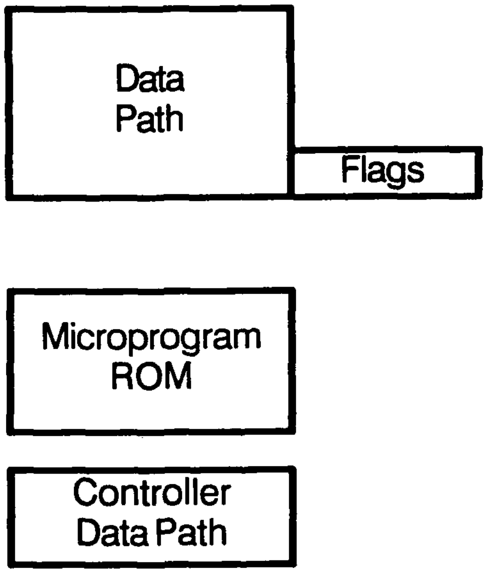


Figure 6.4 Second Alternate Controller Floor Plan

multiple calls of the layout-data-path defstruct in the data-path.l program.

2. Microprogram ROM Floor Plan

The internal floor plan for the microprogram ROM is shown in Figure 6.5. The ROM array is located on the right of the microprogram ROM and individual bit signals may be connected to the top or bottom of each bit column. The rows for the first processes are located at the top of the ROM array followed by the rows of the second process. The rows of the last process are followed by the rows of the first conditional group. The rows of the first conditional group are followed by the rows of the next conditional group. The row for the first state (state number = 0) is at the top of the rows for a process and the row for the last state is at the bottom of the rows for a process. The row for the first alternative of a conditional group is at the top of the rows for the conditional group and the row for the last

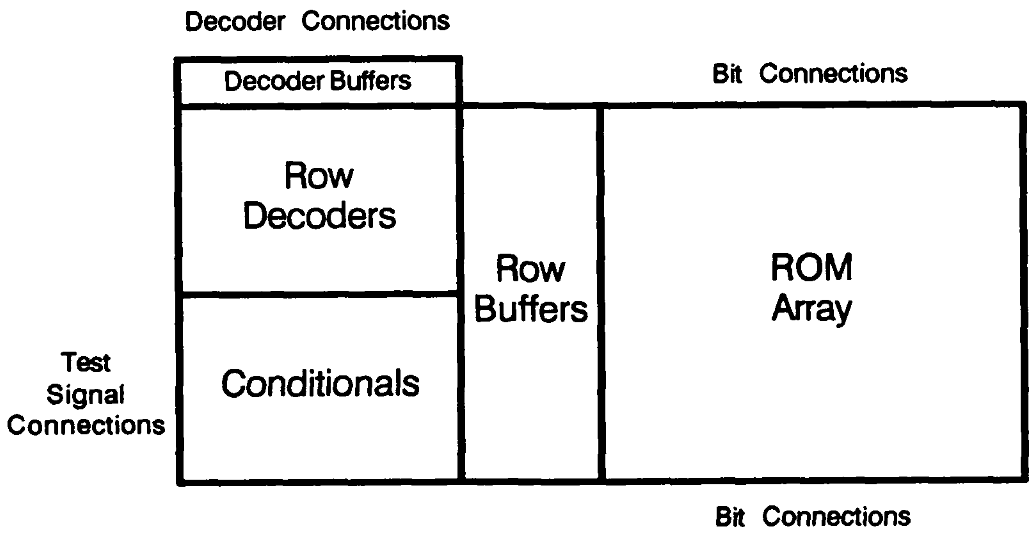


Figure 6.5 Microprogram ROM Floor Plan

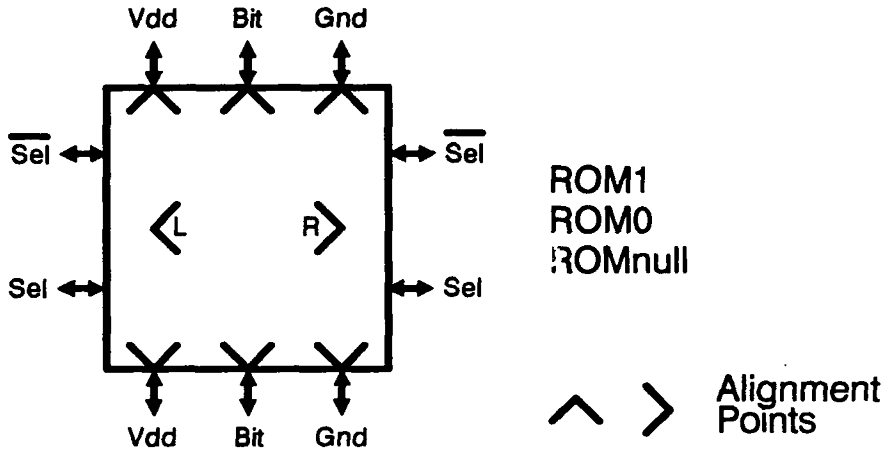
alternative of the conditional group is at the bottom of the rows for the conditional group. Row decoders for all processes are located at the top left of the microprogram ROM. Conditional units are located at the bottom left of the microprogram ROM. Connections for the test signals for the conditional units are made on the left of the microprogram ROM. Buffer units are located between the row decoders and ROM array, between the conditionals and the ROM array, and above the row decoders. Connections to the buffer units for the row decoders are made on the top or left of the microprogram ROM. All internal connections in the microprogram ROM unit are made by cell overlap or abutment. There is no internal routing in the microprogram ROM.

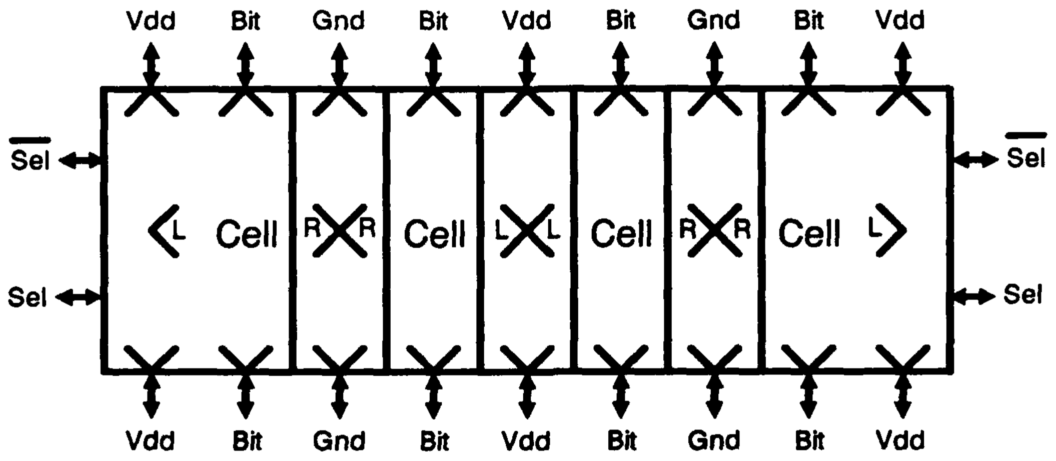
D. CELL LOGIC FUNCTIONS

In this section, the microprogram ROM units are divided into cells and the logic function of each cell is determined. This section contains technology independent functional descriptions. The actual circuits contained in these cells for the SCMOS technology are presented in Section VI.E.

1. ROM Array Cells

The ROM array is constructed from three basic cells that generate a logic one, a logic zero, or a high impedance on the vertical bit line. The cells are named ROM1, ROM0, and ROMnull. The configurations of the cells and the way they join to form rows are shown in Figure 6.6. In all of the cell figures, cell placement is controlled by the left, right, top, and bottom (L R T B) alignment points. Normally, a left alignment point matches a right alignment





Row of Four Cells

Figure 6.6 ROM Array Cells

point in an adjacent cell and a top alignment point matches a bottom alignment point. The ROM array cells are the exception to this convention. Since alternate columns of cells in the ROM array are mirrored about the vertical axis, right alignment points are aligned with right alignment points and left alignment points are aligned with left alignment points. The bottom part of Figure 6.6 shows four ROM array cells that have been joined to form a row. Since the alignment points are inside the cells, there is a lot of overlap. The arrows outside the cell indicate possible data or power flow directions. If an arrow outside the cell matches a alignment point inside the cell, then the alignment point may used for connections outside the microprogram ROM. All connections inside the microprogram ROM are performed by cell abutment or overlap.

a. ROM1 Cell

The ROM1 cell generates a logic one on the vertical bit line when it receives row selection; when it is not selected, the cell produces a high impedance on the bit line. The cell passes the bit line, power, and ground vertically and passes row selection horizontally. The cell has left and right alignment points. Alternate columns are mirrored about the vertical axis to permit running power and ground along the left and right edges. The ROM1 cell has signal alignment points for power, ground, and the bit line on the top and bottom of the cell.

b. ROM0 Cell

The ROM0 cell generates a logic zero on the vertical bit line when it receives row selection; when it is not selected, the cell produces a high impedance on the bit line. The cell passes the bit line, power, and ground vertically and passes row selection horizontally. The cell has left and right alignment points. Alternate columns are mirrored about the vertical axis to permit running power and ground along the left and right edges. The ROM0 cell has signal alignment points for power, ground, and the bit line on the top and bottom of the cell.

c. ROMnull Cell

The ROMnull cell has no connection between the signal on the vertical bit line and row selection. The cell passes the bit line, power, and ground vertically and passes row selection horizontally. The cell has left and right alignment points. Alternate columns are mirrored about the vertical axis to permit running power and ground along the left and right edges. The ROMnull cell has signal alignment points for power, ground, and the bit line on the top and bottom of the cell.

2. <u>Row Decoder Cells</u>

The row decoder cells select a single row of the microprogram ROM based on a binary state number. The row decoder cells are DecodeLSB, DecodeEven0, DecodeEven1, DecodeOdd0, DecodeOdd1, EvenInverter, OddInverter, EvenConnect, OddConnect, and Two-State. The configuration of the cells that form a row decoder for a sixteen state process is in Figure 6.7. A cell is not required below the bottom DecodeEven1 cell in the Bit 2 column. This DecodeEven1 cell drives both the DecodeOdd0 and DecodeOdd1 cells at the bottom of the Bit 1 column.

a. DecodeLSB Cell

The DecodeLSB cell decodes the least significant bit (LSB) of the binary state number. This cell is shown in Figure 6.8. This cell controls two rows of the ROM array. This arrangement allows the size of the DecodeLSB cell to match the small vertical dimension of the ROM array cells. If the DecodeLSB receives an enable signal from the cell on its left, it selects one of the two ROM array rows. If the LSB is zero, the top row is selected. If the LSB is one, the bottom row is selected. The cell passes the LSB, the complement of the LSB, power, and ground vertically. The cell receives an enable signal on the left and generates two row selections on the right. The cell has left and right alignment points. The right alignment point coincides with the left alignment point of the top RowBuffer on its right.

b. DecodeEven0 and DecodeEven1 Cells

The DecodeEven0 and DecodeEven1 cells decode even bits of the binary state number. These cells are shown in Figure 6.9. If the DecodeEven0 cell receives an enable signal from the cell on its left and the even bit of the

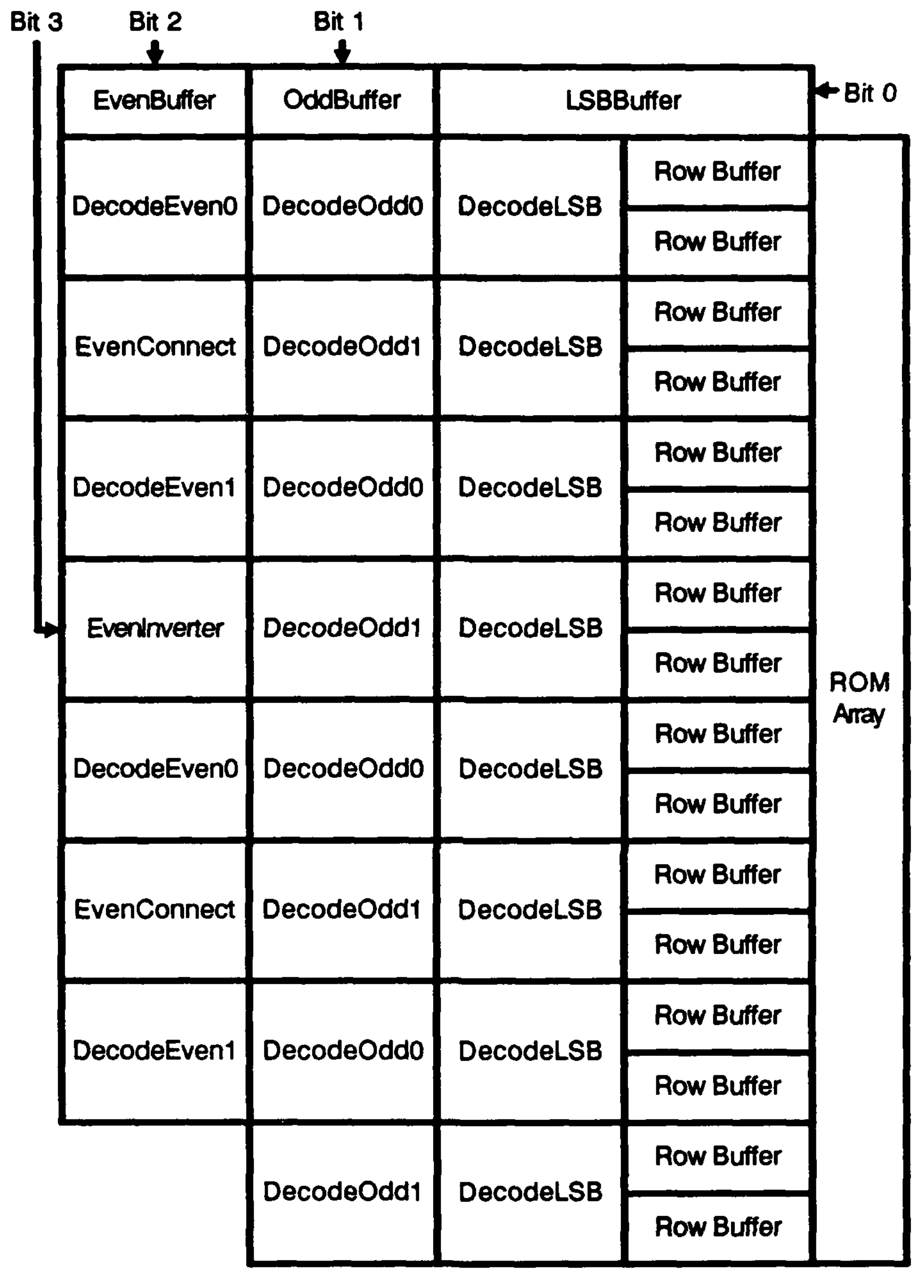


Figure 6.7 Row Decoder

binary state number associated with it is zero, it enables the cell on its right. If the DecodeEven1 cell receives an

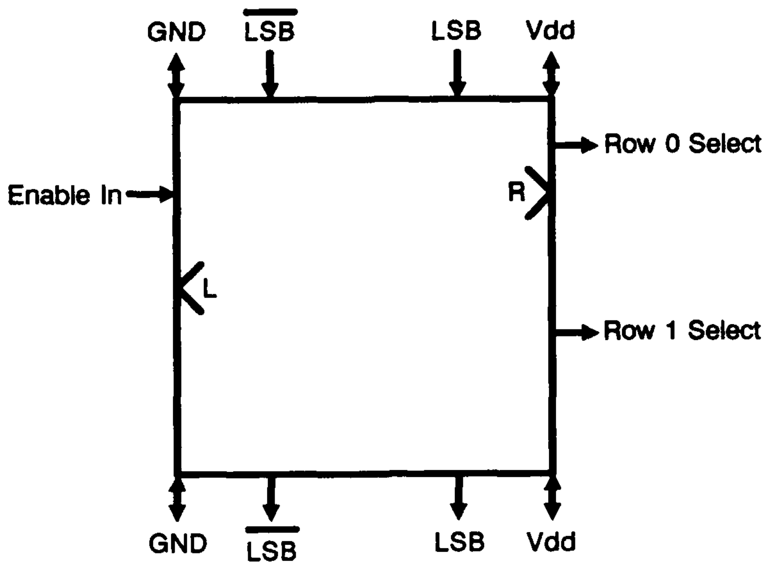
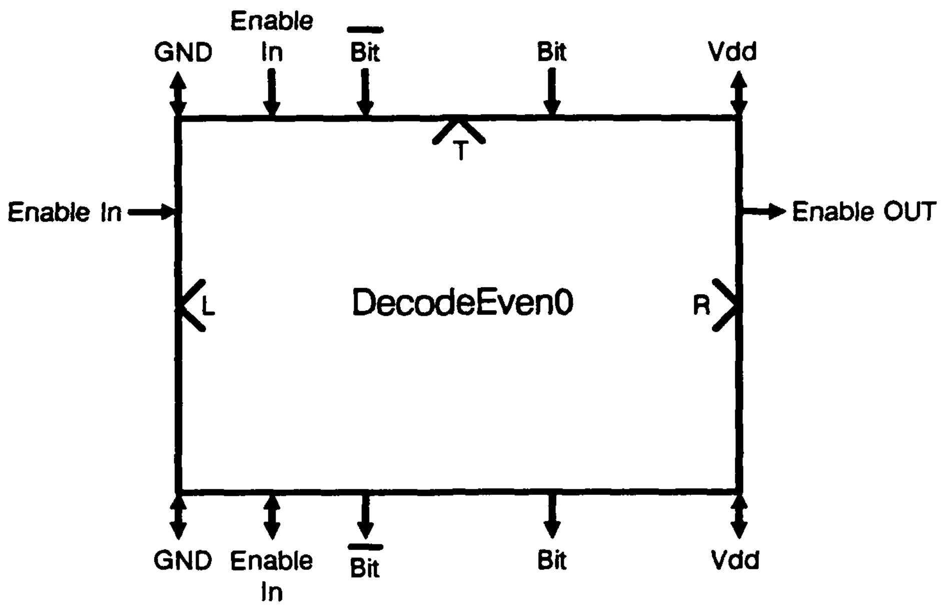


Figure 6.8 DecodeLSB Cell

enable signal from the DecodeEven0 cell above it and the even bit of the binary state number associated with it is one, it enables the cell on its right. Both cells pass the even bit of the binary state number, the complement of the even bit of the binary state number, power, and ground vertically. The DecodeEven0 cell receives an enable signal on the left and generates an enable on the right. The DecodeEven0 also transmits the enable signal it receives on the left to the DecodeEven1 cell below it. The DecodeEven1 cell receives an enable signal from the DecodeEven0 cell above it and generates an enable on the right. If the cells are in the left column of the row decoder, the enable for the two cells may be received from above the DecodeEven0



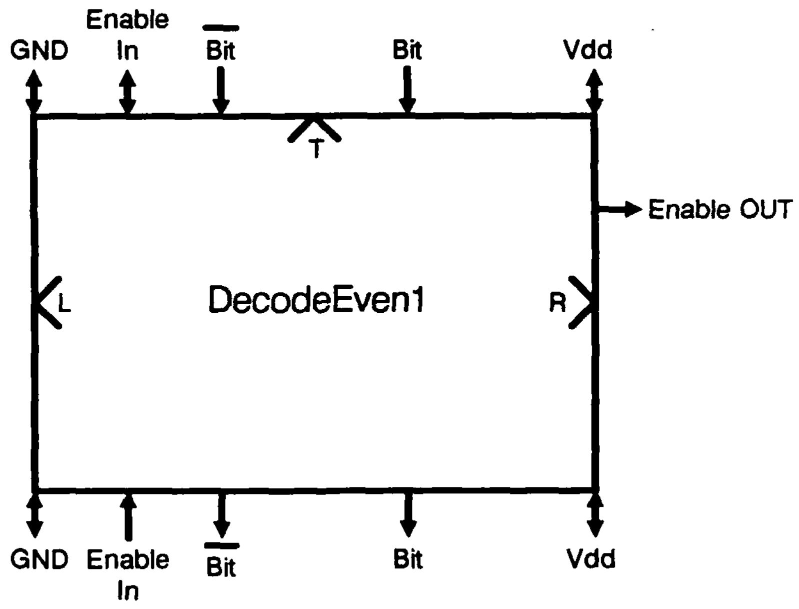


Figure 6.9 DecodeEven0 and DecodeEven1 Cells

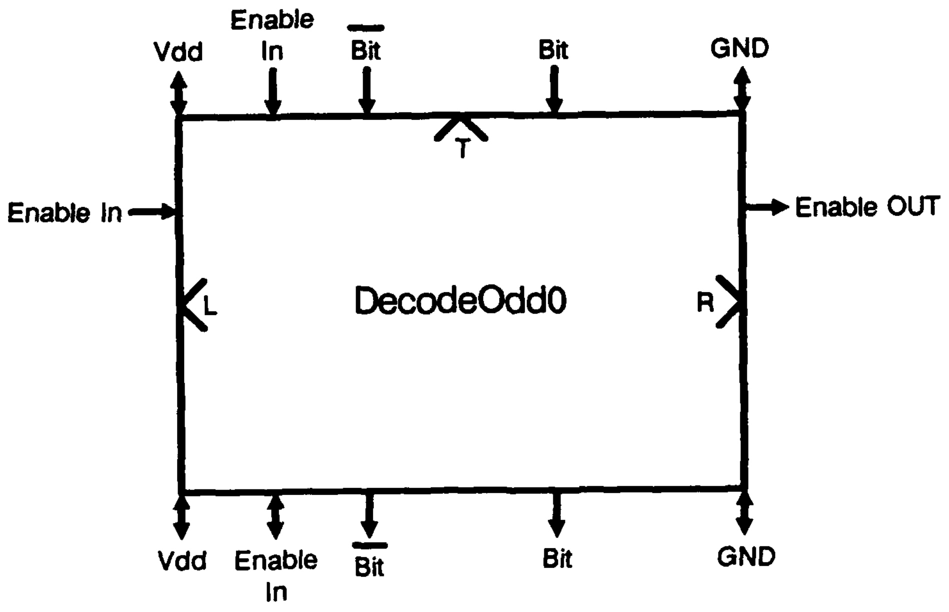
cell or below the DecodeEven1 cell. The cells have left, right, and top alignment points.

c. DecodeOdd0 and DecodeOdd1 Cells

The DecodeOdd0 and DecodeOdd1 cells decode odd bits of the binary state number. These cells are shown in Figure 6.10. If the DecodeOdd0 cell receives an enable signal from the cell on its left and the odd bit of the binary state number associated with it is zero, it enables the cell on its right. If the DecodeOdd1 cell receives an enable signal from the DecodeOdd0 cell above it and the odd bit of the binary state number associated with it is one, it enables the cell on its right. Both cells pass the odd bit of the binary state number, the complement of the odd bit of the binary state number, power, and ground vertically. The DecodeOdd0 cell receives an enable signal on the left and generates an enable on the right. The DecodeOdd0 also transmits the enable signal it receives on the left to the DecodeOdd1 cell below it. The DecodeOdd1 cell receives an enable signal from the DecodeOdd0 cell above it and generates an enable on the right. If the cells are for the bit that is one less than the most significant bit (MSB), the enable for the two cells may be received from the cell above the DecodeOdd0 cell or below the DecodeOdd1 cell. The cells have left, right, and top alignment points.

d. EvenInverter and OddInverter Cells

The EvenInverter and OddInverter cells are used to connect the most significant bit (MSB) to the row decoders. These cells are shown in Figure 6.11. The



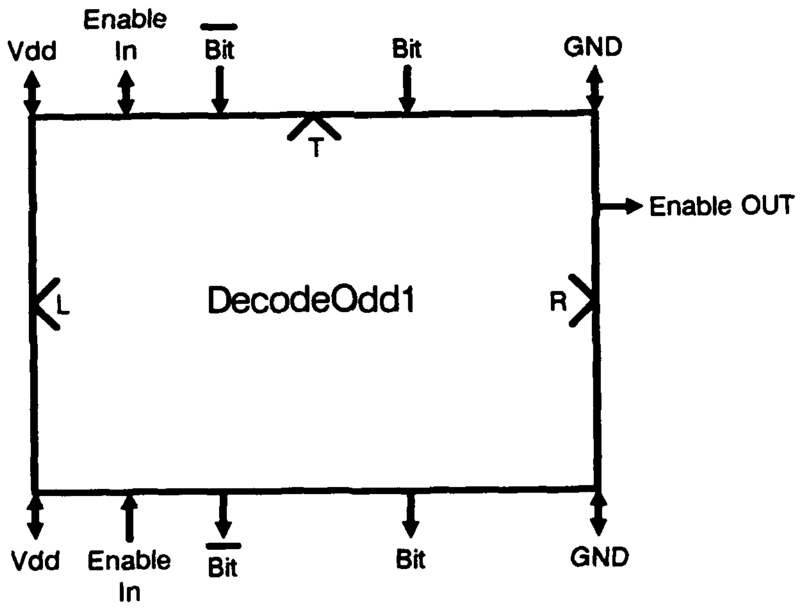
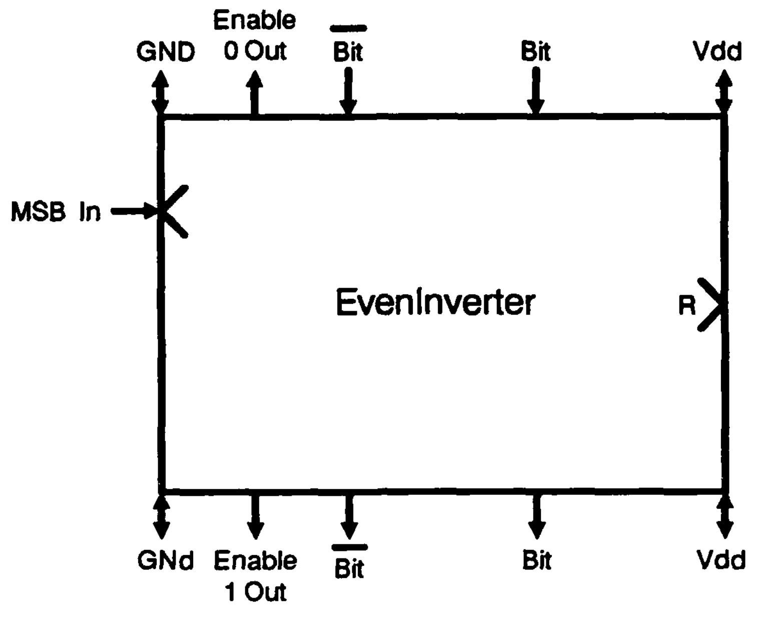


Figure 6.10 DecodeOdd0 and DecodeOdd1 Cells

EvenInverter has one set of DecodeEven0 and DecodeEven1 cells above it and one set below it. The OddInverter has one set of DecodeOdd0 and DecodeOdd1 cells above it and one



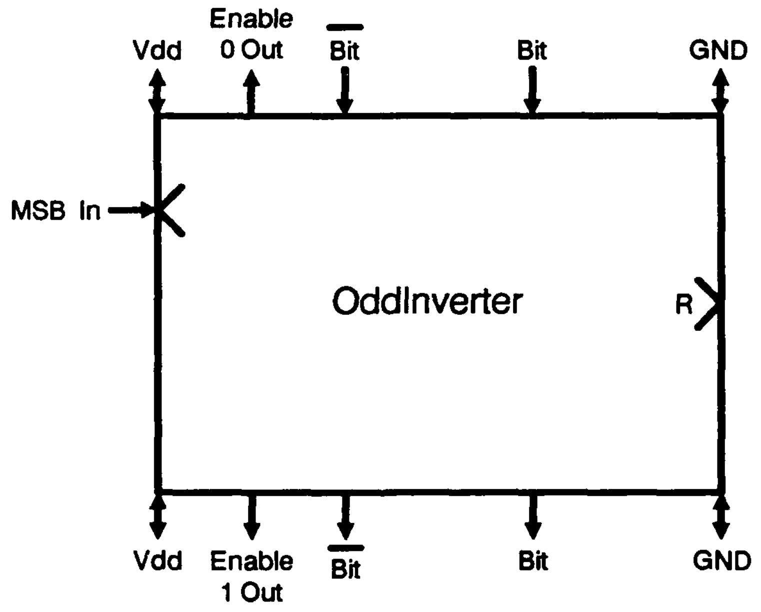


Figure 6.11 EvenInverter and OddInverter Cells

set below it. If the number of states is not an even power of two and less than 1.5 times the largest power of two in the number of states, the lower set of DecodeEven or

DecodeOdd cells may be replaced by an inverter. The EvenInverter receives an odd MSB on the left and the OddInverter receives an even MSB on the left. If the MSB is zero, the EvenInverter or OddInverter enables the cells above it. If the MSB is one, the EvenInverter or OddInverter enables the cells below it. The EvenInverter cell pass the even bit of the binary state number, the complement of the even bit of the binary state number, power, and ground vertically. The OddInverter cell pass the odd bit of the binary state number, the complement of the odd bit of the binary state number, power, and ground vertically. Both cells have an alignment point on the right and a signal alignment point for the MSB on the left.

e. EvenConnect and OddConnect Cells

The EvenConnect cell is used to connect DecodeEven0, DecodeEven1, and EvenInverter cells that do not abut vertically. The OddConnect cell is used to connect DecodeOdd0, DecodeOdd1, and OddInverter cells that do not abut vertically. These cells are shown in Figure 6.12. Both cells pass a bit of the binary state number, the complement of a bit of the binary state number, cell enable, power, and ground vertically. Both cells have alignment points on the left and right.

f. Two-State Cell

The Two-State cell is used for row decoding for processes that have only two states. This cell is shown in

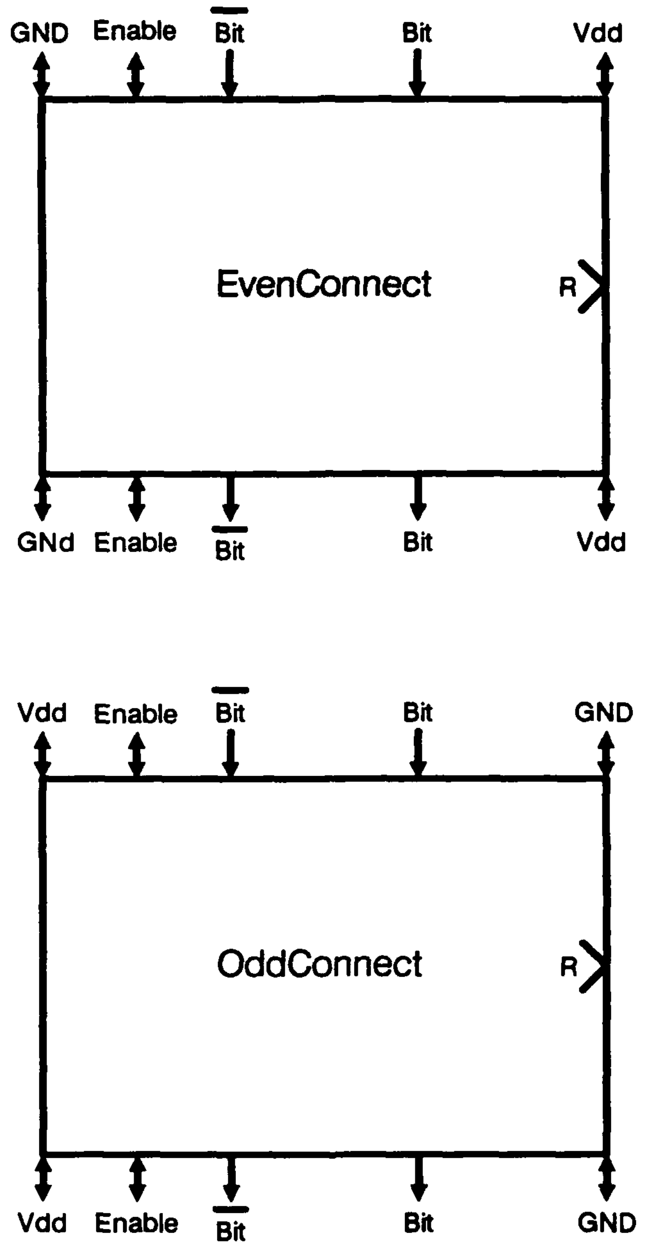


Figure 6.12 EvenConnect and OddConnect Cells

Figure 6.13. This cell controls two rows of the ROM array. This arrangement allows the size of the Two-State cell to

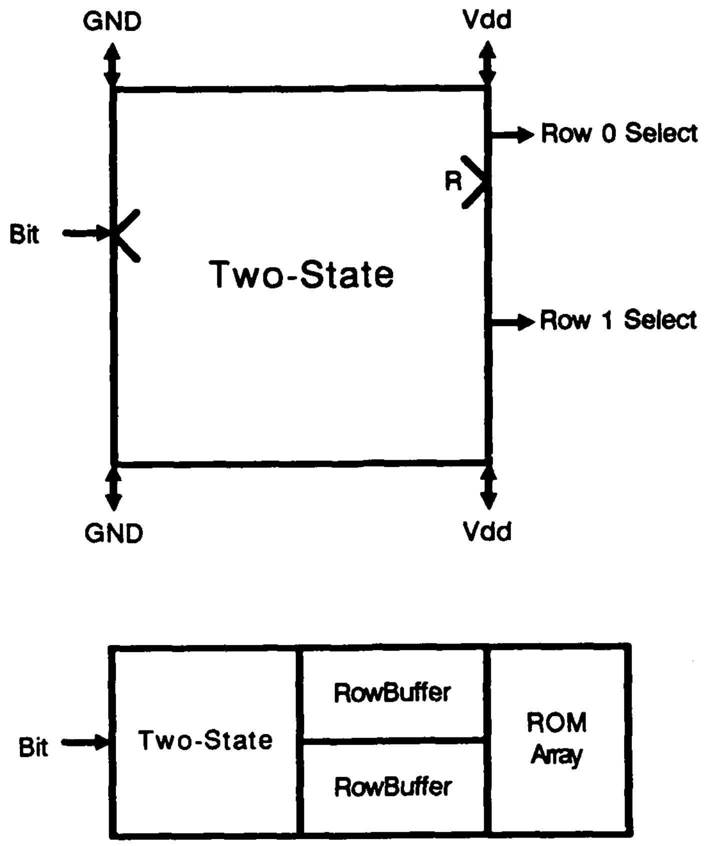


Figure 6.13 Two-State Cell

match the small vertical dimension of the ROM array cells. The cell receives a single bit of the binary state number on the left. If the single bit of the binary state number is zero, it selects the top row of the two ROM array rows. If the single bit of the binary state number is one, it selects the bottom row of the two ROM array rows. The cell passes power and ground vertically. The cell has an alignment point on the right and a signal alignment for the single bit of the binary state number on the left. The right alignment point coincides with the left alignment point of the top RowBuffer on its right.

3. Conditional Cells

Conditional cells are arranged in groups with one group for each cond statement in the design specification. The configuration of the conditional cells for the following segment of a design specification is shown in Figure 6.14:

```
(cond (test1 (setq a b))
      (t (setq a c)))
(cond (test2 (setq b c))
      (test3 (setq b a))
      (t (setq b a)))
```

If the conditional group is enabled, it may select one of the rows in the ROM array. The row selected is determined

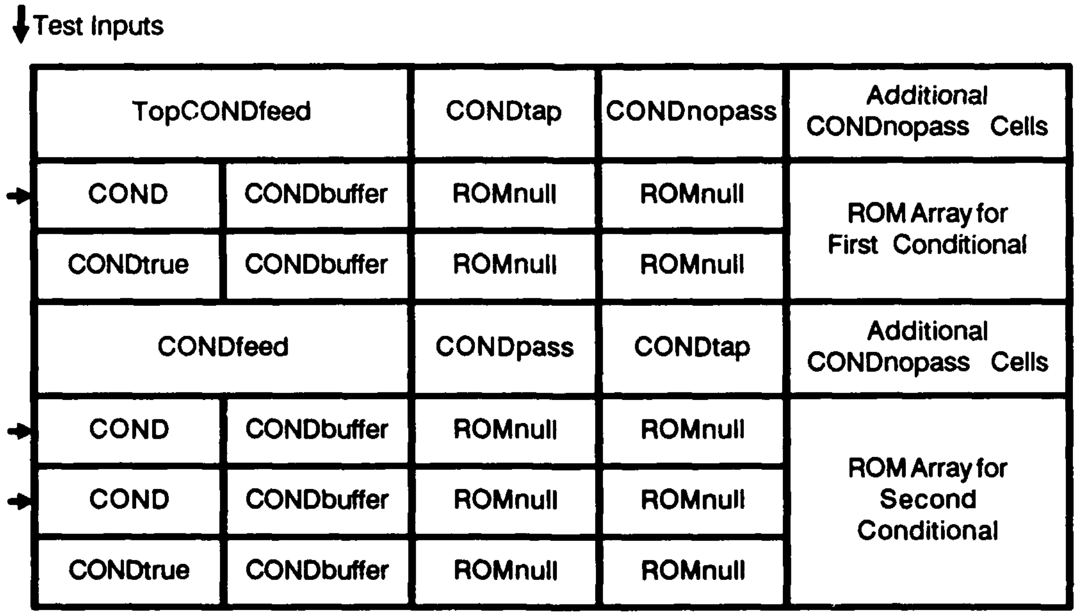


Figure 6.14 Conditional Cells

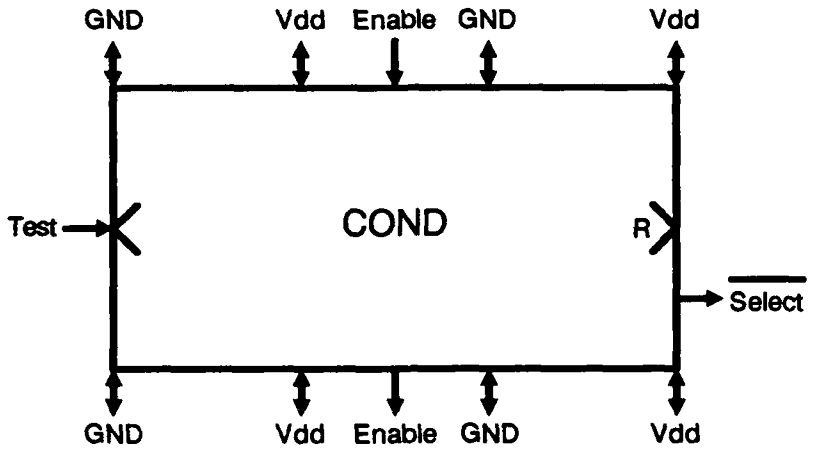
by the test signals connected to the conditional cells and the configuration of the conditional cells. The conditional cells are COND, CONDtrue, CONDfeed, topCONDfeed, CONDtap, CONDpass, and CONDnopass.

a. COND Cell

The COND cell is used to select a row of the ROM array based on enable and test inputs. This cell is shown in Figure 6.15. The COND cell receives an enable from the cell above it and a test signal on the left. If the COND cell is enabled and the test signal is true, the COND cell selects the cell on the right. If the COND cell is enabled and the test signal is false, the COND cell enables the cell below it. The cell passes power and ground vertically. The cell has an alignment point on the right and a signal alignment for the test signal on the left.

b. CONDtrue Cell

The CONDtrue cell is used to select a row of the ROM array based on an enable input. This cell is shown in Figure 6.15. The CONDtrue cell receives an enable from the cell above it. If the CONDtrue cell is enabled, the CONDtrue cell selects the cell on the right. The cell passes power and ground vertically. The cell has an alignment point on the right.



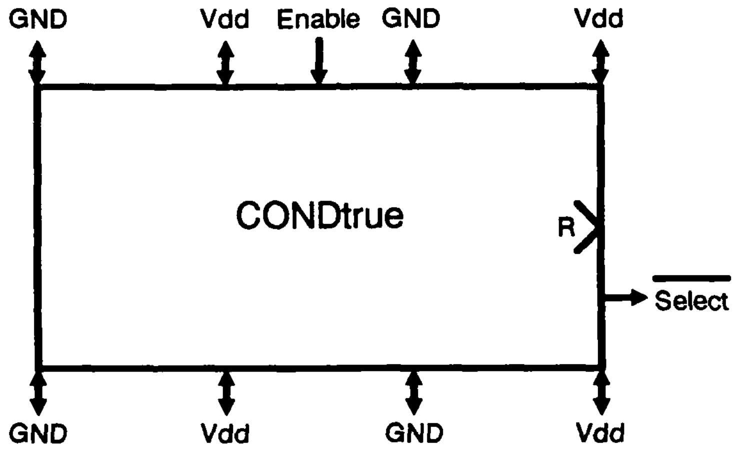
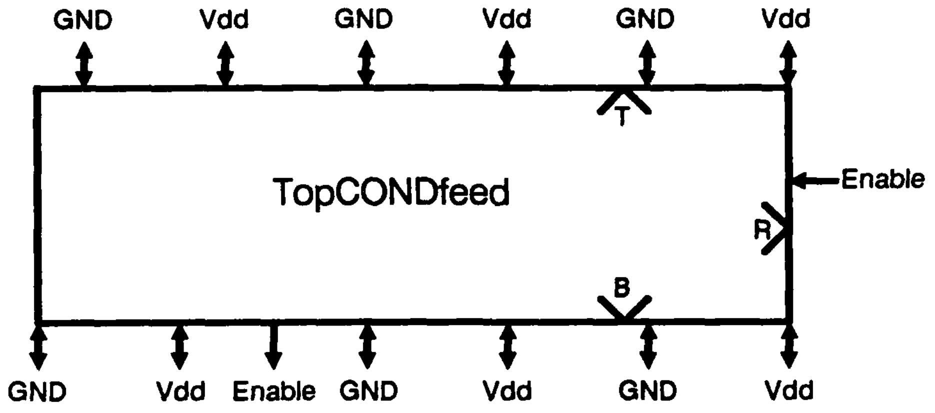


Figure 6.15 COND and CONDtrue Cells

c. CONDfeed and TopCONDfeed Cells

The CONDfeed and topCONDfeed cells form the top of conditional cell groups. These cells are shown in Figure 6.16. They transmit an enable from the CONDtap or CONDpass cell on the right to the COND cell below the topCONDfeed or CONDfeed cell. The CONDfeed cell is used if there is another conditional group above it and the topCONDfeed cell is



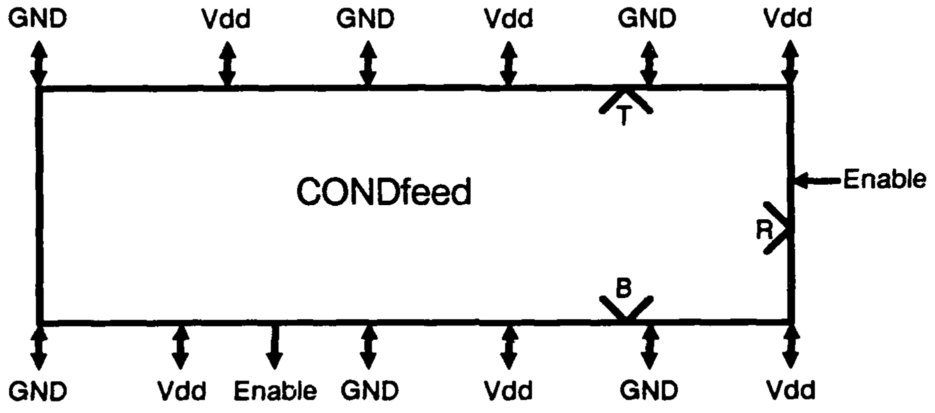


Figure 6.16 TopCONDfeed and CONDfeed Cells

used if there is a row decoder above it. The cells pass power and ground vertically. Both cells have alignment points on the top, right, and bottom. The top and bottom alignment points determine the spacing of the entire horizontal row of the microprogram ROM.

d. CONDtap Cell

The CONDtap cell extracts a bit from the ROM array and transmits it to the cell on the left. The value of the bit is controlled by ROM0 and ROM1 cells located in the column above or below the CONDtap cell. This cell is shown in Figure 6.17. The cell passes power, ground, and a bit line vertically. The cell has right and left alignment points.

e. CONDpass Cell

The CONDpass cell transmits a bit from the cell on the right to a cell on the left. This cell is shown in Figure 6.17. The cell passes power, ground, and a bit line vertically. The cell has right and left alignment points.

f. CONDnopass Cells

The CONDnopass cell passes power, ground, and a bit line vertically. This cell is shown in Figure 6.17. The cell has right and left alignment points.

4. Buffer Cells

The buffer cells generate complementary outputs for single inputs. The configurations of units using the buffer cells are shown in Figures 6.7, 6.13, and 6.14. The drive current on the output of the buffers may be much greater than the input current of the buffer. The buffer cells are RowBuffer, CONDbuffer, LSBbuffer, EvenBuffer, and OddBuffer.

a. RowBuffer Cell

The RowBuffer cell forms the buffer between the row decoders and the ROM array. The cell is shown in Figure 6.18. The cell receives a select on the left and provides complementary select outputs with greater drive current on

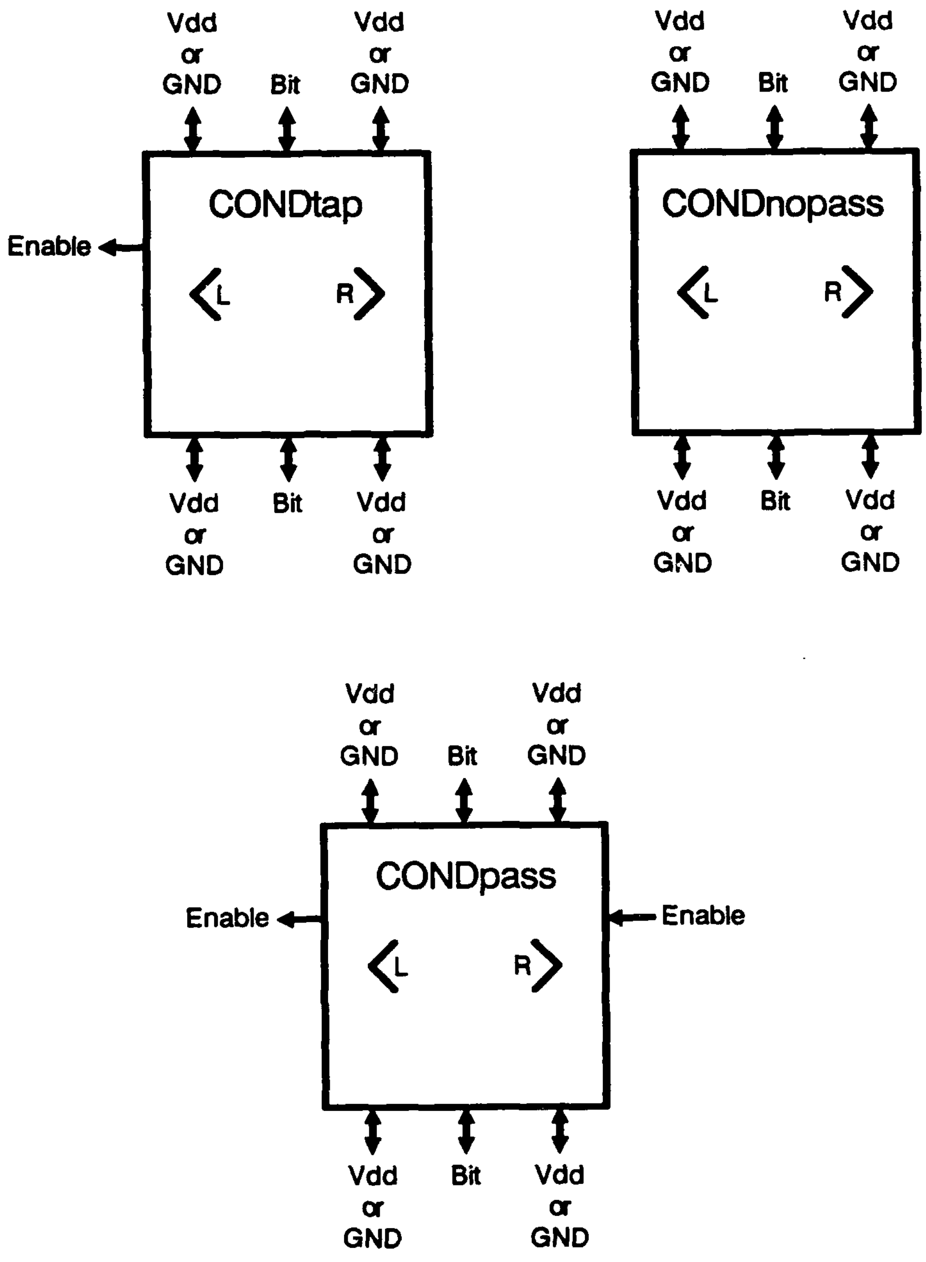
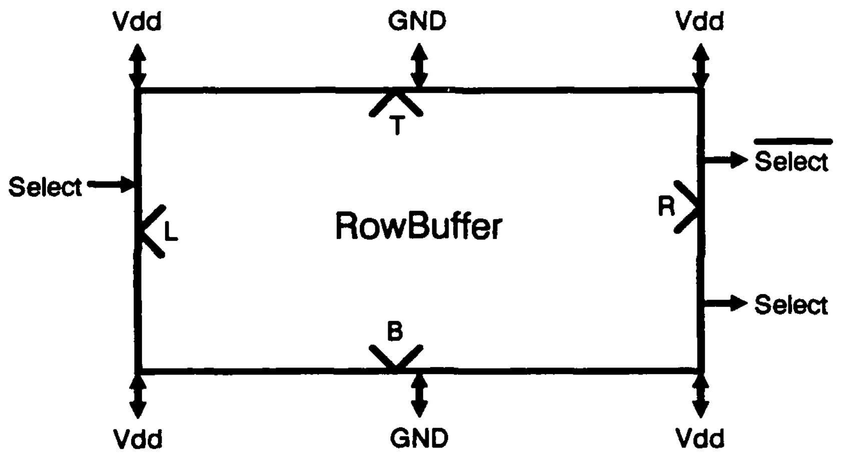


Figure 6.17 CONDtap, CONDpass, and CONDnopass Cells

the right. The cell passes power and ground vertically. The cell has l∂ft, right, top, and bottom alignment points.



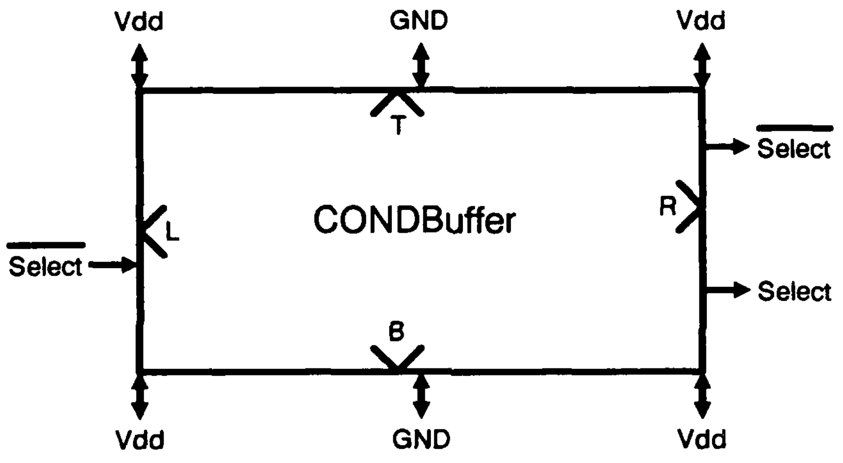


Figure 6.18 RowBuffer and CONDbuffer Cells

The top and bottom alignment points determine the spacing of the entire horizontal row of the microprogram ROM.

b. CONDbuffer Cell

The CONDbuffer cell forms the buffer between the conditional cells and the ROM array. The cell is shown in Figure 6.18. The cell receives the complement of a select on the left and provides complementary select outputs with greater drive current on the right. The cell passes power and ground vertically. The cell has left, right, top, and bottom alignment points. The top and bottom alignment points determine the spacing of the entire horizontal row of the microprogram ROM.

c. LSBbuffer Cell

The LSBbuffer cell provides complementary bit signals for the DecodeLSB cells. This cell is shown in Figure 6.19. The cell receives a least significant bit (LSB) input on the right and produces complementary bit outputs on the bottom. The cell passes power and ground vertically. The cell has an alignment point on the bottom and a signal alignment point on the right for the LSB.

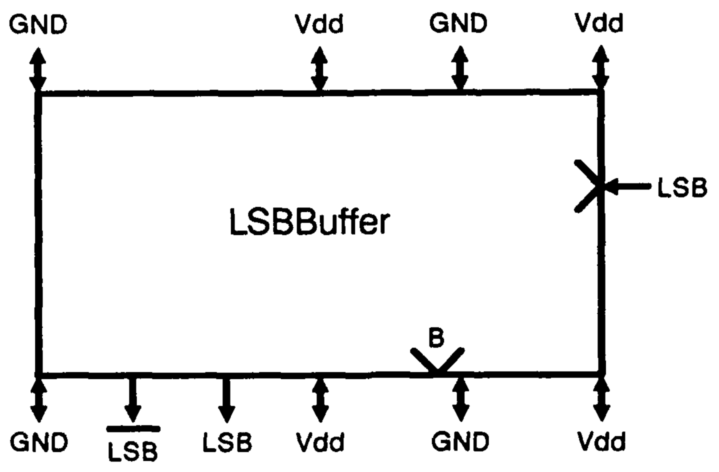


Figure 6.19 LSBbuffer Cell

d. EvenBuffer Cell

The EvenBuffer cell provides complementary bit signals for the DecodeEven0 and DecodeEven1 cells. This cell is shown in Figure 6.20. The cell receives an even bit of the binary state number as input on the top and produces complementary bit outputs on the bottom. The cell passes power and ground vertically. The cell has an alignment point on the bottom and a signal alignment point on the top for the even bit.

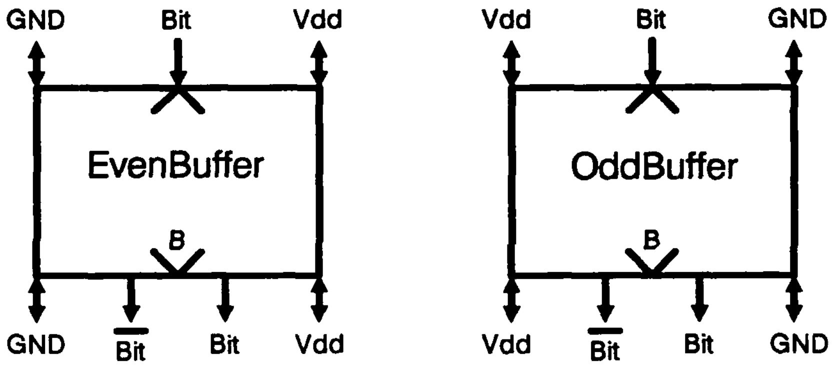


Figure 6.20 EvenBuffer and OddBuffer Cells

e. OddBuffer Cell

The OddBuffer cell provides complementary bit signals for the DecodeOdd0 and DecodeOdd1 cells. This cell is shown in Figure 6.20. The cell receives an odd bit of the binary state number as input on the top and produces complementary bit outputs on the bottom. The cell passes power and ground vertically. The cell has an alignment point on the bottom and a signal alignment point on the top for the odd bit.

E. CELL INTERNAL STRUCTURE

In this section, the internal structure of each cell in the microprogram ROM is determined.

1. <u>ROM Array Cells</u>

The ROM array is constructed from three basic cells that generate a logic one, a logic zero, or a high impedance on the vertical bit line. The cells are named ROM1, ROM0, and ROMnull.

a. ROM1 Cell

The ROM1 cell has power, ground, and bit line routed vertically in metal1. The power and ground in metal1 inside the ROM array connect to metal2 power and ground buses outside the ROM array using metal2 contacts. Thus, the bit lines are able to pass under the buses. The select signal and the complement of the select signal are routed horizontally in metal2. The ROM1 cell contains a p-type transistor connected between power and the signal line with the gate connected to the complement of the select signal as shown in Figure 6.21.

b. ROM0 Cell

The ROM0 cell has power, ground, and bit line routed vertically in metal1. The select signal and the complement of the select signal are routed horizontally in

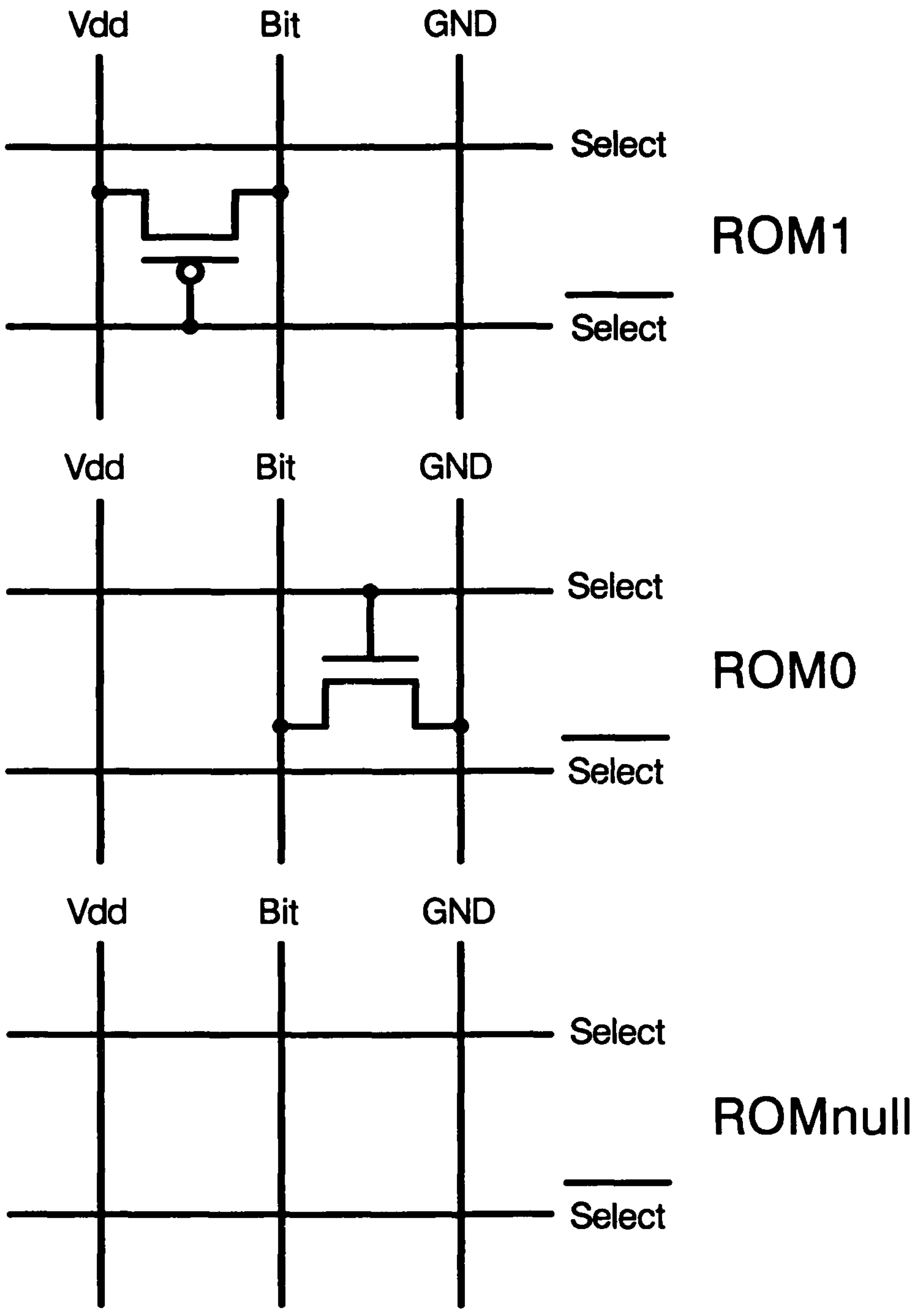

Vdd
Bit
GND
Select
ROM1
$\overline{\text{Select}}$
Vdd
Bit
GND
Select
ROM0
$\overline{\text{Select}}$
Vdd
Bit
GND
Select
ROMnull
$\overline{\text{Select}}$

Figure 6.21 ROM Array Cells

metal2. The ROM0 cell contains a n-type transistor connected between ground and the signal line with the gate connected to the select signal as shown in Figure 6.21.

c. ROMnull Cell

The ROMnull cell has power, ground, and bit line routed vertically in metal1. The select signal and the complement of the select signal are routed horizontally in metal2. The ROMnull cell is shown in Figure 6.21.

2. Row Decoder Cells

The row decoder cells select a single row of the microprogram ROM based on a binary state number. The row decoder cells are DecodeLSB, DecodeEven0, DecodeEven1, DecodeOdd0, DecodeOdd1, EvenInverter, OddInverter, EvenConnect, OddConnect, and Two-State. To simplify the circuitry, the decoders for even bits use NOR gates and the decoders for odd bits use NAND gates. The outputs of the NOR gates are active high and the inputs to the NOR gates are active low. The outputs of the NAND gates are active low and the inputs to the NAND gates are active high. The logic configuration of the decoders for the top two rows of the row decoder shown in Figure 6.7 is illustrated in Figure 6.22. The input for the most significant bit comes from an EvenInverter cell.

a. DecodeLSB, DecodeEven0, and DecodeEven1 Cells

The DecodeLSB cell contains two NOR gates and the DecodeEven0 and DecodeEven1 cells contain single NOR

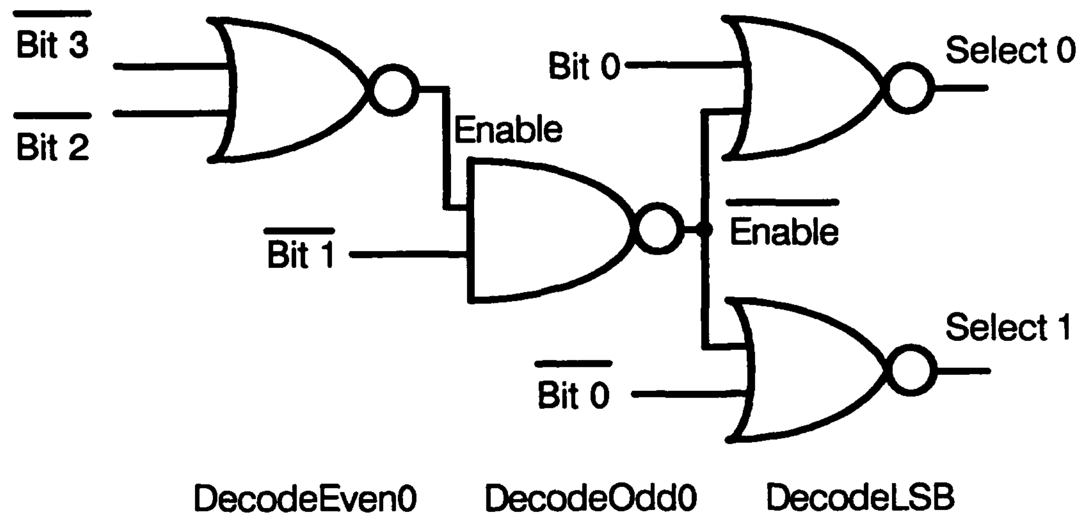

$\overline{\text{Bit 3}}$
$\overline{\text{Bit 2}}$
Enable
$\overline{\text{Bit 1}}$
Bit 0
Select 0
$\overline{\text{Enable}}$
Select 1
$\overline{\text{Bit 0}}$
DecodeEven0
DecodeOdd0
DecodeLSB

Figure 6.22 Row Decode Logic

gates. The logic configurations of these cells is shown in Figure 6.23. An SCMOS NOR gate is shown in Figure 6.24. The enable signals for these cells are active low so an enable is a logic zero. The signal(s) output by the gates are active high so a high is a logic one. In the DecodeLSB cell, one of the inputs of each NOR gate is connected to the line for the inverse enable signal. The other input of the top gate is connected to the LSB line and the other input of the bottom gate is connected to the line for the complement of the LSB. One of the inputs of the NOR gate in the DecodeEven0 and DecodeEven1 cells is connected to the enable input. The other input of the NOR gate in the DecodeEven0 cell is connected to the bit line. The other input of the NOR gate in the DecodeEven1 cell is connected to the line

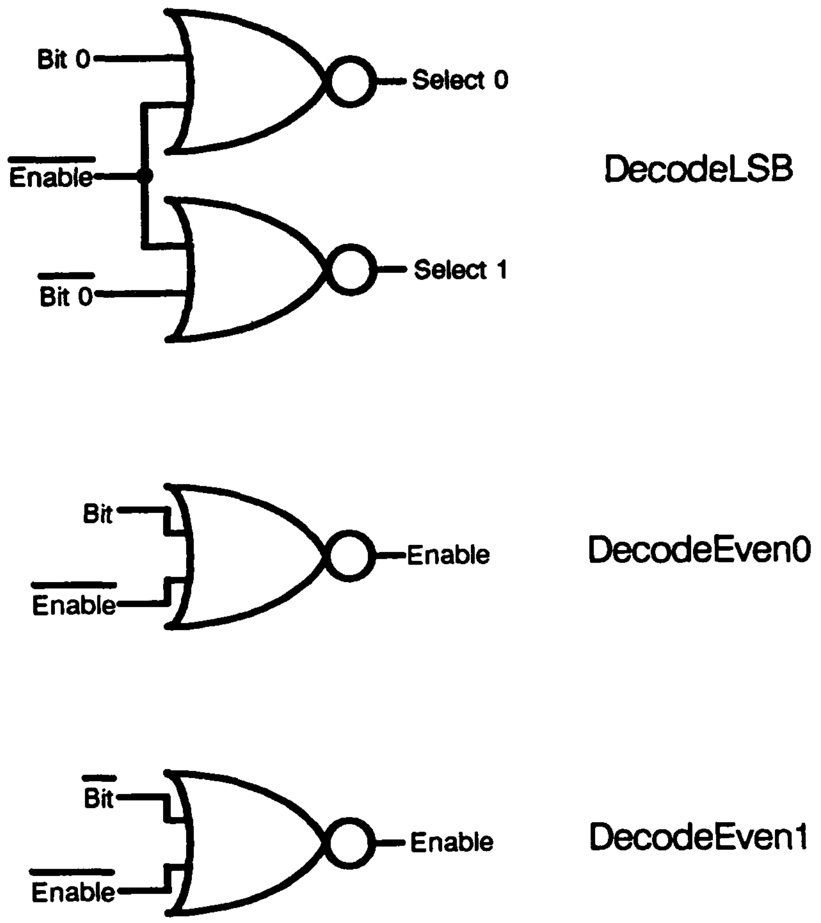

Bit 0
Select 0
$\overline{\text{Enable}}$
DecodeLSB
Select 1
$\overline{\text{Bit 0}}$
Bit
Enable
$\overline{\text{Enable}}$
DecodeEven0
$\overline{\text{Bit}}$
Enable
$\overline{\text{Enable}}$
DecodeEven1

Figure 6.23 DecodeLSB, DecodeEven0, and DecodeEven1 Logic

for the complement of the even bit. Power, ground, the bit line, and the line for the complement of the bit are routed vertically in metal1. The enable and select signals are routed in Metal2.

b. DecodeOdd0 and DecodeOdd1 Cells

The DecodeEven0 and DecodeEven1 cells contain single NAND gates. The logic configurations of these cells are shown in Figure 6.25. An SCMOS NAND gate is shown in

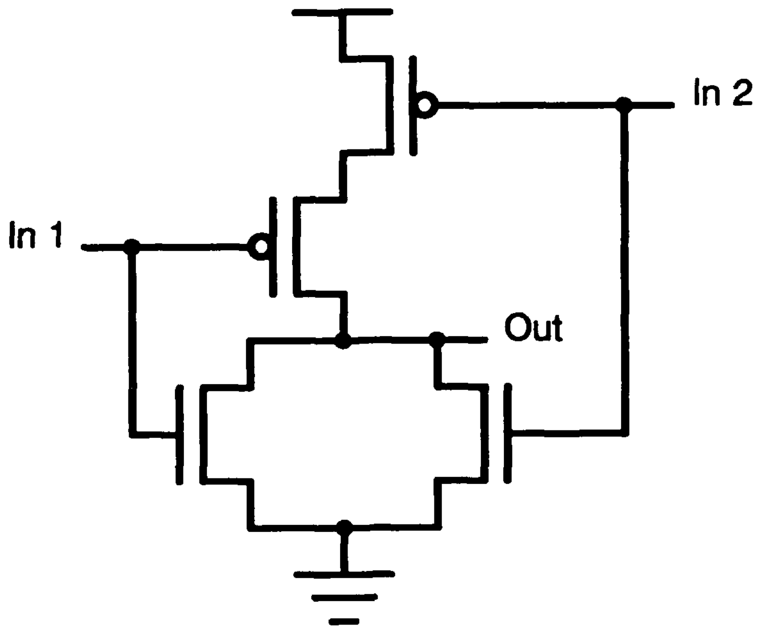


Figure 6.24 NOR Gate

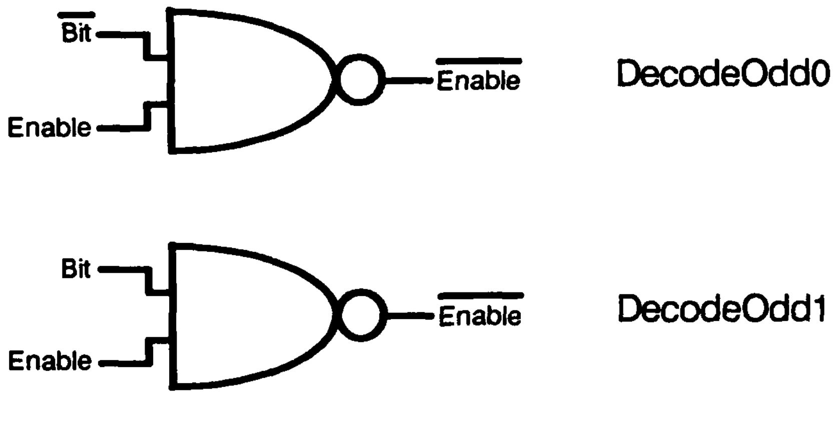


Figure 6.25 DecodeOdd0 and DecodeOdd1 Logic

Figure 6.26. The enable signal for these cells is active high so an enable is a logic one. The enable signal output

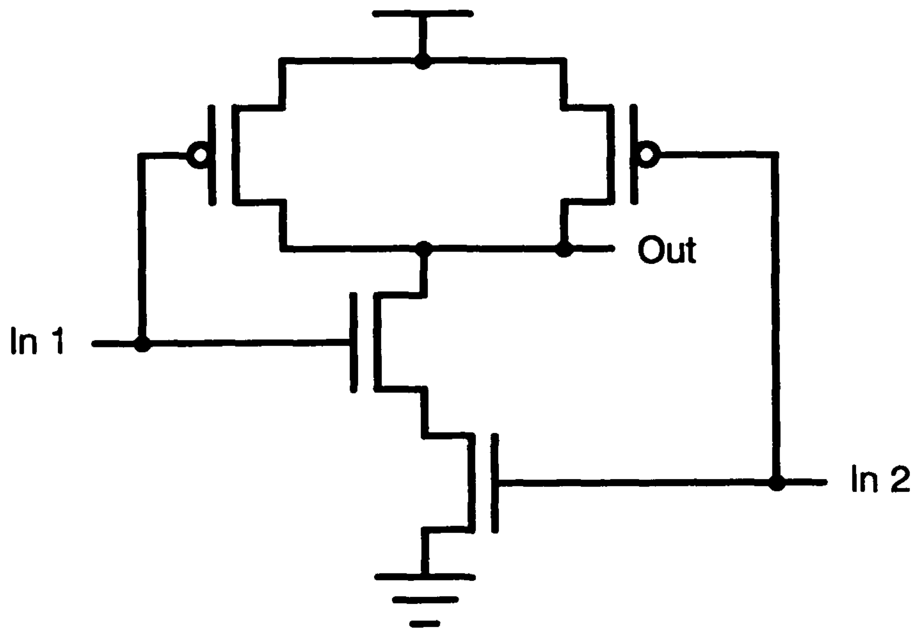


Figure 6.26 NAND Gate

by the cells is active low so an enable is a logic zero. One of the inputs of the NAND gate in the DecodeOdd0 and DecodeOdd1 cells is connected to the enable input. The other input of the NAND gate in the DecodeOdd0 cell is connected to the line for the complement of the odd bit line. The other input of the NAND gate in the DecodeOdd1 cell is connected to the odd bit line. Power, ground, the bit line, and the line for the complement of the bit are routed vertically in metal1. The enable and select signals are routed in Metal2.

c. EvenInverter and OddInverter Cells

The EvenInverter and OddInverter cells contain single inverters. The logic configurations of these cells are shown in Figure 6.27. An SCMOS inverter gate is shown in Figure 6.28. The most significant bit (MSB) is routed to the input of the inverter. The MSB is also routed to the enable input of the top pair of DecodeEven0 and DecodeEven1 cells for the EvenInverter. The MSB is also routed to the enable input of the bottom pair of DecodeOdd0 and DecodeOdd1 cells for the OddInverter. The output of the inverter is routed to the enable input of the bottom pair of DecodeEven0 and DecodeEven1 cells for the EvenInverter. The output of the inverter is routed to the enable input of the top pair

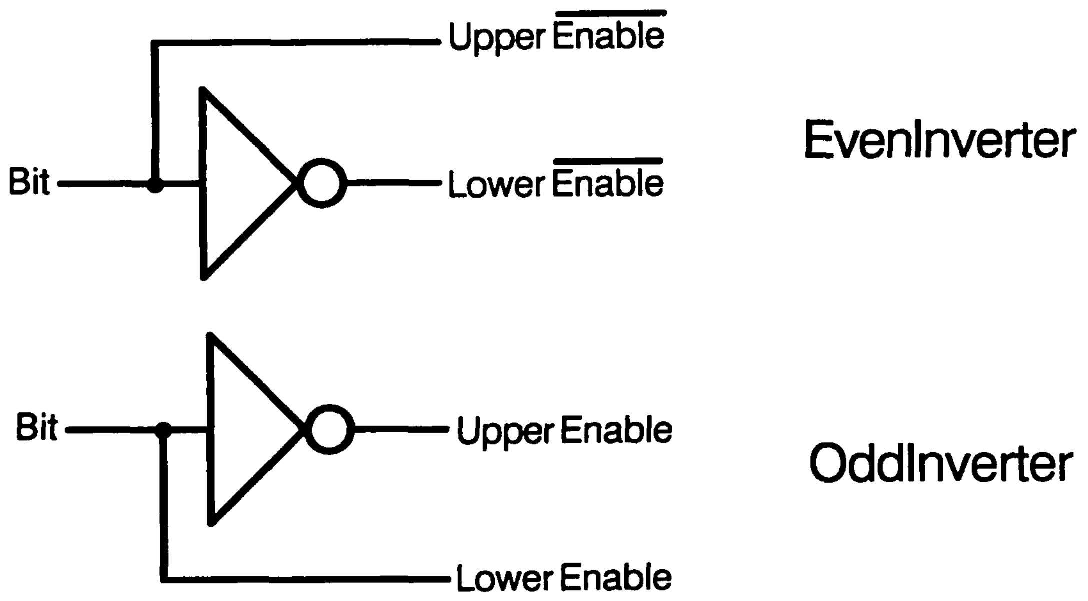

Upper $\overline{\text{Enable}}$
Bit
Lower $\overline{\text{Enable}}$
EvenInverter
Bit
Upper Enable
OddInverter
Lower Enable

Figure 6.27 EvenInverter and OddInverter Logic

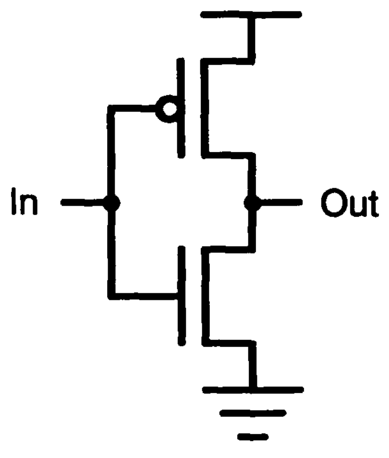


Figure 6.28 Inverter

of DecodeOdd0 and DecodeOdd1 cells for the OddInverter. Power, ground, the bit line, and the line for the complement of the bit are routed vertically in metal1. The MSB and enable signals are routed in Metal2.

d. EvenConnect and OddConnect Cells

The EvenConnect and OddConnect cells contain metal routing. Power, ground, the bit line, and the line for the complement of the bit are routed vertically in metal1. The enable signals are routed vertically in Metal2.

e. Two-State Cell

The Two-State cell contains a single inverter. The logic configuration of the cell is shown in Figure 6.29. An SCMOS inverter gate is shown in Figure 6.28. The single bit of the binary state number is routed to the input of the inverter. The single bit of the binary state number is also

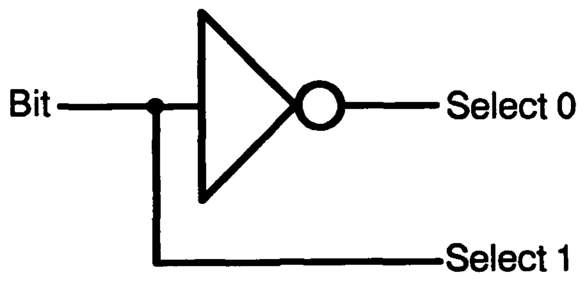


Figure 6.29 Two-State Logic

routed to the select input of the bottom RowBuffer connected to the Two-State cell. The output of the inverter is routed to the select input of the top RowBuffer connected to the Two-State cell. Power and ground are routed vertically in metal1. The single bit of the binary state number and select signals are routed in Metal2.

3. <u>Conditional Cells</u>

Conditional cells are arranged in groups with one group for each cond statement in the design specification. If the conditional group is enabled, it may select one of the rows in the ROM array. The row selected is determined by the test signals connected to the conditional cells and the configuration of the conditional cells. The conditional cells are COND, CONDtrue, CONDfeed, topCONDfeed, CONDtap, CONDpass, and CONDnopass.

a. COND Cell

The COND cell contains an inverter, a NAND gate, and a NOR gate. These gates are shown in Figures 6.24, 6.26, and 6.28. The logical configuration of the COND cell is shown in Figure 6.30. The enable input, test input, and enable output are active high so a high is a logical one. The select output is active low so a low is a logical one. The enable input is connected to the input of the inverter and one input of the NAND gate. The test input is connected to one input of the NOR gate and one input of the NAND gate. The output of the inverter is connected to one input of the NOR gate. The output of the NAND gate is connected to the select input of the CONDbuffer and the output of the NOR gate is connected to the enable output. Power, ground, and the enable signals are routed vertically in metal1 or polysilicon. The test input and select signals are routed in Metal2.

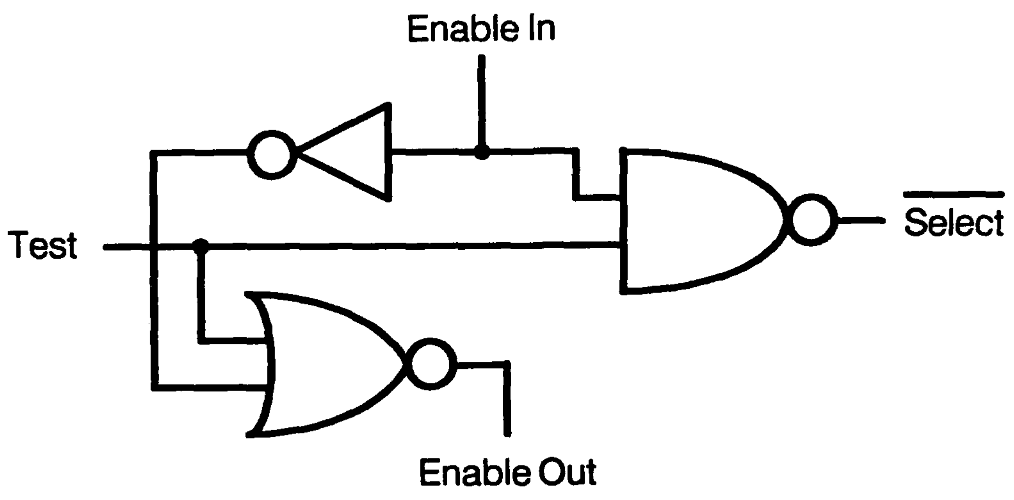


Figure 6.30 COND Logic

b. CONDtrue Cell

The CONDtrue cell contains a single inverter. The logic configuration of this cell is shown in Figure 6.31. An SCMOS inverter gate is shown in Figure 6.9. The enable input is routed to the input of the inverter. The output of the inverter is routed to the select input of a CONDbuffer. Power, ground, and the enable signal are routed vertically in metal1 or polysilicon. The select signal is routed in Metal2.

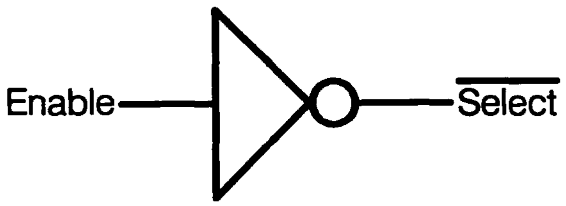


Figure 6.31 CONDtrue Logic

c. CONDfeed, TopCONDfeed, CONDtap, CONDpass, and CONDnopass Cells

The CONDfeed, TopCONDfeed, CONDtap, CONDpass, and CONDnopass cells route power, ground, and bit lines vertically in metal1 and enable lines horizontally in metal2.

4. <u>Buffer Cells</u>

The buffer cells generate complementary outputs for single inputs. The drive current on the output of the buffers may be much greater than the input current of the buffer. The buffer cells are RowBuffer, CONDbuffer, LSBbuffer, EvenBuffer, and OddBuffer.

a. RowBuffer Cell

The RowBuffer cell contains two inverters that are configured as superbuffers. The logic configuration of this cell is shown in Figure 6.32. An SCMOS inverter gate is shown in Figure 6.28. The select input is connected to the input of the first inverter. The output of the first inverter is connected to the input of the second inverter and to the line for the complement of the row select signal. The output of the second inverter is connected to the line for the row select signal. Power and ground are routed vertically in metal1. The select signals are routed in Metal2.

b. CONDbuffer Cell

The CONDbuffer cell contains two inverters that are configured as superbuffers. The logic configuration of this cell is shown in Figure 6.32. An SCMOS inverter gate is shown in Figure 6.28. The select input is connected to the input of the first inverter. The output of the first inverter is connected to the input of the second inverter and to the line for the row select signal. The output of the second inverter is connected to the line for the complement of the row select signal. Power and ground are routed vertically in metal1. The select signals are routed in Metal2.

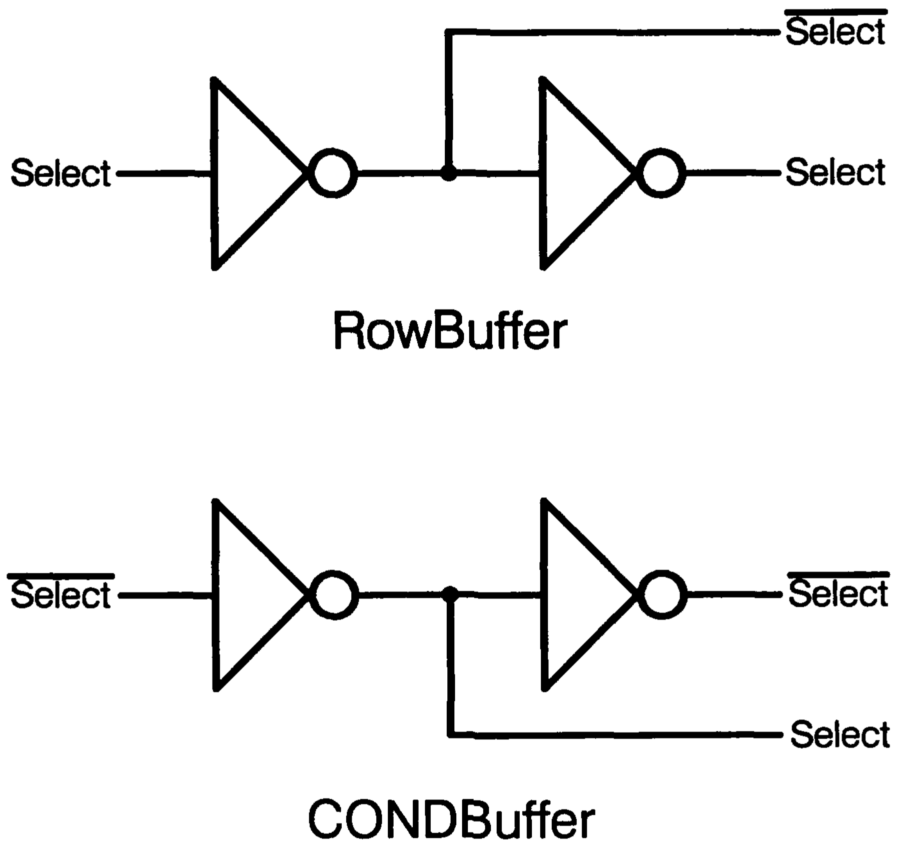

$\overline{\text{Select}}$
Select
Select
RowBuffer
$\overline{\text{Select}}$
$\overline{\text{Select}}$
Select
CONDBuffer

Figure 6.32 RowBuffer and CONDbuffer Logic

c. LSBbuffer Cell

The LSBbuffer cell contains two inverters that are configured as superbuffers. The logic configuration of this cell is shown in Figure 6.33. An SCMOS inverter gate is shown in Figure 6.28. The LSB input is connected to the input of the first inverter. The output of the first inverter is connected to the input of the second inverter and to the line for the complement of the LSB signal. The output of the second inverter is connected to the line for

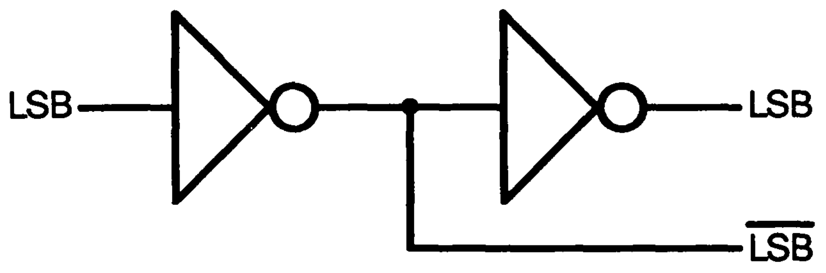


Figure 6.33 LSBbuffer Logic

the LSB signal. Power, ground, and the output signals are routed vertically in metal1. The LSB input signal is routed in Metal2.

d. EvenBuffer and OddBuffer Cells

The EvenBuffer and OddBuffer cells contain single inverters. The logic configuration of these cells is shown in Figure 6.34. An SCMOS inverter gate is shown in Figure 6.28. The bit input is routed to the input of the inverter and to the bit line output. The output of the inverter is routed to the line for the complement of the bit.

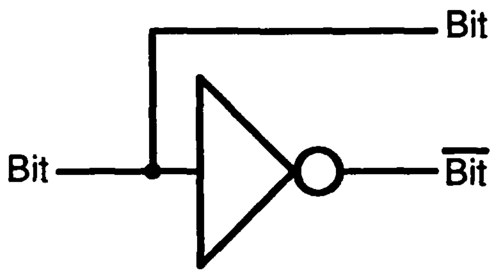


Figure 6.34 EvenBuffer and OddBuffer Logic

Power, ground, and the bit input signal are routed in metal1. The output signals are routed in Metal2.

F. DESIGN VERIFICATION

The microprogram controller design satisfies the controller requirements and conforms to the controller goals. The control signals generated by the controller are functions of the test signals from the data path and the current state numbers of the sequencers for the processes in the design specification. This makes the controller a Mealy FSM. The controller contains units that perform all of the functions shown in Figure 2.9 for the MSC FSM. The growth of the size of the controller is based on linear functions of parameters of the design specification. The controller has a regular structure. All cells in the microprogram ROM connect by abutment or overlap. The circuitry inside the cells of the microprogram ROM is simple for speed and compact size. All cells contain alignment points for easy assembly by a silicon compiler. The design verification reveals no discrepancies between the design and the controller goals and requirements.

VII. CONTROLLER IMPLEMENTATION

Implementation of the microprogrammed controller for MSC involves many interrelated tasks including: installation of the MSC software on the ISI workstations, upgrade of MSC for SCMOS technology and compatibility with the Magic[1] VLSI layout editor, cell layout and test, addition of cells to MSC, and extension of MSC to use the new cells. Addition of the cells to MSC and extension of MSC for the new cells are not completed in this thesis and will require further research.

A. MSC SOFTWARE INSTALLATION

Implementation of the microprogrammed controller for MSC started with installation of the MacPitts software on a clustered set of three ISI workstations. During the process of installation, MacPitts was updated to use the Franz Lisp interpreter Opus 38.78 and the Liszt compiler Opus 8.36 with BSD 4.2 UNIX. During the installation, many errors were corrected to produce error and warning free compilations. Additional modifications were performed to permit use of the Franz Lisp trace, stepping, and debugging programs. A log of all actions taken during installation is contained in Appendix A.

[1]The Magic VLSI layout editor is described in 1986 VLSI Tools: Still More Works by the Original Artists (UCB/CSD 86/272, December 1985).

B. MSC MAGIC AND SCMOS UPGRADE

The CIF output produced by the original MacPitts silicon compiler was not compatible with the Magic VLSI layout editor. Since Magic is the primary VLSI layout editor used at the Naval Postgraduate School, the silicon compiler had to be modified to work with Magic. An additional benefit of Magic compatibility is that the Magic program can extract simulation information for use by the ESIM[2] event driven switch level simulator and the CRYSTAL[3] VLSI timing analyzer. The modifications to MacPitts to produce Magic CIF are in Software Change Proposal SCP-1. SCP-1 is superseded by SCP-2 and is not included in this thesis.

The original version of MacPitts supports a single four or five micron nMOS technology. By adding SCMOS layers to the nMOS layers in MacPitts, a CIF file may be produced that has both SCMOS and nMOS layers. However, this hybrid technology is not suitable for simulation or fabrication. Also, use of this technique to add new technologies to the silicon compiler would require a complete new silicon compiler program for each technology. A more flexible solution is to add command line options to the silicon compiler to select

[2]The ESIM event driven switch level VLSI simulator is described in 1986 VLSI Tools: Still More Works by the Original Artists (UCB/CSD 86/272, December 1985).

[3]The CRYSTAL VLSI timing analyzer is described in 1986 VLSI Tools: Still More Works by the Original Artists (UCB/CSD 86/272, December 1985).

the technology to be used for CIF output. The modifications to MacPitts to add the required command line options is in Software Change Proposal SCP-2. SCP-2 supersedes SCP-1 and is presented in Appendix 2. SCP-2 was approved and incorporated in the MSC baseline. The modifications in SCP-2 enable establishment of an integrated silicon compiler development environment that includes the MSC silicon compiler including an internal functional simulator and Franz Lisp stepping, tracing, and debugging programs, the Magic VLSI layout editor, the ESIM event driven switch level simulator, and the CRYSTAL VLSI timing analyzer.

C. CELL LAYOUT AND TEST

All cells described in Chapter VI have been laid out using the Magic VLSI layout editor. Each cell passes the internal Magic design rule checker. An extraction file for each cell has been produced by using the Magic extract command. The extraction files have been converted into simulation files by the ext2sim[4] program. Each simulation file was tested with the CRYSTAL static circuit check and ESIM event driven switch level simulator to verify proper performance. A test microprogram ROM named ROMtest.mag contains all of the Chapter VI cells with all abutments, overlaps, and orientations that are possible for the microprogram ROM

[4]The ext2sim program is described in _1986 VLSI Tools: Still More Works by the Original Artists_ (UCB/CSD 86/272, December 1985).

floor plan in Figure 6.5. The combinations of cells in ROMtest.mag also pass the internal Magic design rule checker. The simulation file ROMtest.sim is also produced by using the Magic extract command to produce extraction files for each cell in ROMtest.mag and merging these extraction files using the ext2sim program. ROMtest.sim passes the CRYSTAL static electrical check and produces critical paths that correspond to the intended operation of the circuit. Using the ESIM event driven switch level simulator, ROMtest.sim produces correct outputs for all inputs. Proper operation of each cell and combination of cells is assured by this simulation.

D. ADDITION OF CELLS TO MSC

Defsymbols named layout-rom0 and layout-rom1 for the ROM0 and ROM1 cells are contained in the file controller.l. This file has been loaded into a test version of MSC and used to generate test layouts. The test layouts have been loaded into Magic and pass the internal design rule checker. Addition of the remainder of the cells to MSC is not completed in this thesis and will require further research.

The defsymbols were produced manually from the original Magic layouts. This same procedure could be used to add the remainder of the cells into MSC. Another method to load the cells into MSC would be to write a program that generates a defsymbol definition from a Magic or cif file. This process has been used in other thesis research to enter cells into MSC. The final result of either procedure is a large program that is technology dependent. To add a new feature or new technology to MSC using these procedures will require major revision of the MSC program.

A better way to add new features, cells, or technologies to MSC is to develop a mechanism in MSC to use cells that are not part of the MSC program. One way to implement this mechanism involves two new programs.

One program can be written in C or any other language to convert a cif file into a form that is easily read by the Franz Lisp interpreter. This program would produce a file that is formatted as a lisp list. This file would contain the bounding box of the cell, all required labels for the cell, and all layout information for the cell. All measurements would be converted to lambda units so that MSC can scale them to the appropriate minimum feature size. The position of the cell would be translated to a home position. The lisp formatted files would be stored in separate directories for each technology.

The second program would be an additional macro to MSC named cifsymbol that is similar to the defsymbol macro. Each cell used by the compiler would be defined by a cifsymbol definition. The cifsymbol contains the name of the cell and the names of any required labels in the cell. The cifsymbol does not contain any technology information. If the cifsymbol is not used, it is not expanded and the program stays small. The first time the cifsymbol is called, the macro is expanded. When the macro is expanded, a set of directories specified for the MSC technology is searched for the lisp file corresponding to the cell name. When the cell file is found, it is read and immediately output to the --L5-symbol-list file. The cifsymbol call returns a symbol that calls the cell. This symbol does not contain layout information. All subsequent calls to the cifsymbol return similar symbols and do no other processing.

This cifsymbol method will significantly reduce the size of the MSC program, remove technology dependent structures from the MSC program, and increase the efficiency of developing new cells or modifying existing cells that are part of MSC. To replace an existing cell for MSC, a user would simply have a file in a directory earlier in the search path with the same name as the cell being replaced.

E. MSC CONTROLLER EXTENSION

Adding the microprogrammed controller to MSC will require a modification of the object file structure and changes to the extract.l and control.l programs, and the addition of a single bit data-path. These extensions of MSC are not completed in this thesis and will require further research.

1. Object File Modifications

The current nMOS object file structure has five elements for definitions, flags, data-path, control, and pins. The control element contains specifications for a NOR gate based Weinberger control logic array. The microprogrammed controller object file control element should contain specifications for a microprogram ROM. These specifications will include the number of processes, the number of states in each process, the number of cond statements, the number of substates in each cond statement, the sources of test inputs for each cond substate, and the ROM contents for each state and substate.

2. Extract.l Modifications

The functions in the extract.l program that produce the object file control element must be modified to produce the new structure described in Section VII.E.1.

3. Extraction Example

This section provides an example of the functions that must be performed by the modified extract.l to derive the information required for the new object file control element. This example is based on the following taxi cab meter design specification presented in An Introduction to MacPitts (MIT RVLSI-3, pp. 13-29, 10 February 1983).

```
(program taxi 8
 (def 17 power)
 (def 1 ground)
 (def 2 phia)
 (def 3 phib)
```

```
(def 4 phic)
(def timer register)
(def fare register)
(def reset signal input 5)
(def time-on signal input 6)
(def hire signal input 7)
(def mile-mark signal input 8)
(def display port tri-state (9 10 11 12 13 14 15 16))
(def charge-time signal internal)
(def maximum-time constant 100)
(def base-fare constant 20)
(def cost-per-mile constant 50)
(def cost-per-time const 10)
(process time-clock 0
 off
   (cond (time-on (setq timer 0) (go on))
         (t (go off)))
 on
   (cond (time-on (cond ((= timer maximum-time)
                          (setq timer 0)
                          (signal charge-time))
                       (go on))
         (t (setq timer 0) (go off))))
(process fare-clock 0
 for-hire
   (cond (hire (setq fare base-fare) (go hired))
         (t (go for-hire)))
 hired
   (par (cond ((not hire) (go for-hire))
              ((and charge-time mile-mark)
               (setq fare (+ (+ fare cost-per-mile)
                             cost-per-time))
               (go hired))
              (charge-time
               (setq fare (+ fare cost-per-time))
               (go hired))
              (mile-mark
               (setq fare (+ fare cost-per-mile))
               (go hired))
              (t go hired))
        (setq display fare))))
```

There are two processes in the taxi program. Each process will have its own row decoder. The first process is named time-clock and it has two states named off and on. The second process is named fare-clock and it has two states named for-hire and hired. The modified control.l will use Two-State cells for the row decoder for each process.

There are five cond statements in the program. The modified control.l program will layout each cond statement as a cond group in accordance with the design in Section VI.D.3.

The first cond statement has two substates. The test input for the first substate is the input signal time-on. The second substate is always true so there is no input. The modified control.l will use the CONDtrue cell for this substate.

The second cond statement has two substates. The test input for the first substate is the input signal time-on. The second substate is always true so there is no input. The modified control.l will use the CONDtrue cell for this substate.

The third cond statement has two substates. The test input for the first substate is the test line from the = organelle in the data-path. The second substate is always true so there is no input. The modified control.l will use the CONDtrue cell for this substate.

The fourth cond statement has two substates. The test input for the first substate is the input signal hire. The second substate is always true so there is no input. The modified control.l will use the CONDtrue cell for this substate.

The fifth cond statement has five substates. The test input for the first substate is the internal signal from a not organelle in the single bit data-path. The input to this organelle is the external signal hire. The test input for the second substate is the internal signal from an and organelle in the single bit data-path. The inputs to this organelle are the internal signal charge-time and the external signal mile-mark. The test input for the third substate is the internal signal charge-time. The test input for the fourth substate is the input signal mile-mark. The fifth substate is always true so there is no input. The modified control.l will use the CONDtrue cell for this substate.

The following is the object file data-path element generated by the current MSC for the taxi program:

```
((register sequencer-time-clock-state -1
                  (((constant 0)) ((internal 2))))
 (port-internal sequencer-time-clock-next-state -2
                  (((constant 1)) ((constant 0))))
 (bit (0) (((internal 1))))
 (register timer -3 (((constant 0)) ((internal 4))))
 (organelle = 0 (((internal 3) (constant 100))))
 (organelle |1+| -4 (((internal 3))))
 (register sequencer-fare-clock-state -5
                  (((constant 0)) ((internal 6))))
 (port-internal sequencer-fare-clock-next-state -6
                  (((constant 1)) ((constant0))))
 (bit (0) (((internal 5))))
 (port-output display (((internal 7))))
 (register fare -7
          (((constant 20)) ((internal 9)) ((internal 8))))
 (organelle + -8
          (((internal 7 (constant 50))
           ((internal 7) (constant 10))))
 (organelle + -9 (((internal 8) (constant 10)))))
```

Each element in this list corresponds to a unit in the data-path. If the layout of the circuit is oriented so that the data-path is at the top, the first element corresponds to the left unit in the data-path. The data-path for the taxi program has thirteen units. This list can be used to determine the names and the order of the multiplexer control lines and test signal lines between the data path and the controller. The multiplexer control lines will correspond to columns in the microprogram ROM.

The input element in a unit specification determines the input multiplexer configuration of that unit. The input element is the fourth element in register, internal port, and organelle unit specifications and the third element in bit and output port unit specifications. The input element is always a list. If the list has a single element, there is only one set of inputs to the unit that are connected to a degenerate multiplexer and there are no multiplexer control lines. If there is more than one element in the list, the number of elements is the number of multiplexer control lines. Each element of the list in the input element is a list of inputs that are enabled when the control line for this element is high. There are internal and constant inputs. An internal input comes from an internal bus. A constant input is generated by connecting the inputs to the multiplexer to power or ground.

All units may have multiplexer control lines. Register units have an enable control line that determines whether or not the register contents are allowed to be updated during state transition. Bit units have test lines that are used to provide single bit outputs to the controller. Organelle units may have test lines and carry input lines depending on the function of the organelle. The organelle unit test lines and carry lines may not be connected to the controller.

The first unit in the data-path is a register for sequencer-time-clock-state. This register contains the state number for the time-clock process. The -1 for the third element in the register specification states that the output of the register is connected to internal bus 1. The fourth element is a list that has two elements. This means that there are two multiplexer control lines for this unit named (mpx 1 1) and (mpx 1 2). The second element in the multiplexer name is the unit number. The multiplexer control line (mpx 1 1) enables a constant input of 0 for the unit and (mpx 1 2) enables an input from internal bus 2. The left line is (mpx 1 2) and the right line is (mpx 1 1). The (mpx 1 1) control line is connected to the external reset signal to force the contents of the register to 0. This multiplexer selection must override any other multiplexer selections. This would violate the requirement for true multiplexers. A better solution is to use a register that has a reset input. This input would be connected to the reset input signal. To the right of the multiplexer control lines is an enable control line for the register named (control-line 1 1). When this line is high, it enables loading of the register.

The second unit in the data-path is an internal port for sequencer-time-clock-next-state. This port generates the next state numbers for the time-clock process. The -2 for the third element in the internal port specification states that the output of the internal port is connected to internal bus 2. The fourth element is a list that has two elements. This means that there are two multiplexer control lines for this unit named (mpx 2 1) and (mpx 2 2). The second element in the multiplexer name is the unit number. The multiplexer control line (mpx 2 1) enables a constant input of 1 for the unit and (mpx 2 2) enables a constant input of 0. The left line is (mpx 2 2) and the right line is (mpx 2 1). When (mpx 2 1) is high, the next state is on and when (mpx 2 2) is high, the next state is off.

The third unit in the data-path is a bit unit. This unit extracts bit 0 from its input word. The third element is a list that has one elements. This means that this bit unit has a single input source. As a result, there are no multiplexer control lines for this unit. The input is connected directly to internal bus 1. Since this unit is for a single bit, it has a single test line (test-line 3 1) that returns the bit to the controller.

These first three units form a no-counter-no-stack sequencer for the first process.

The fourth unit is a register for the timer. The output of the register is connected to internal bus 1. The register has two multiplexer control lines and an enable control line named (mpx 4 1), (mpx 4 2), and (control-line 4 1).

The fifth unit is an = organelle unit. Since the fourth element of the organelle specification is a list with a single element that is a list of two elements, this unit has two inputs with one signal for each input. The inputs are connected to internal bus 3 and the constant 100. This unit does not have any multiplexer control lines. The unit has a single test line (test-line 5 1) that is connected to the controller. This signal is high when the internal bus 3, the output of the timer register, is equal to 100.

The sixth unit is a 1+ organelle unit. This is an incrementer that has internal bus 4 for output and internal bus 3 for input. This incrementer unit has no multiplexer control lines and a single test line (test-line 6 1). This test line is high when there is a carry out of the most significant bit of the incrementer. This test line is not connected to the controller.

Units seven, eight, and nine form a no-counter-no-stack sequencer for the second process. Their configuration is similar to units one, two, and three. The (mpx 7 1) multiplexer control line is connected to the external reset signal.

Unit ten is a port-output unit for the display output port. The input of this unit is connected to internal bus 7 and the output is connected to lines below the internal bus lines that bring the output of the unit to the left of the data-path and from there to the pads. There are no multiplexer control lines or test lines for this unit.

The eleventh unit is a register for the fare. The output of the register is connected to internal bus 7. The register has three multiplexer control lines and an enable control lines named (mpx 11 1), (mpx 11 2), (mpx 11 3), and (control-line 11 1).

The twelfth unit is a + organelle unit. This organelle unit contains a two input full adder. This unit has two multiplexer control lines. The (mpx 12 1) control line enables the internal bus 7 and constant 50 inputs to the adder. The (mpx 12 2) control line enables the internal bus 7 and constant 10 inputs to the adder. The output of the unit is connected to internal bus 8. The unit has a carry out test line that is not connected to the controller.

The thirteenth unit is a + organelle unit. This organelle unit contains a two input full adder. This unit has a single set of inputs and no multiplexer control lines. The adder inputs are connected to internal bus 8 and constant 10 inputs. The output of the adder is connected to internal bus 9. The unit has a carry out test line that is not connected to the controller.

The columns in the microprogram ROM should be in the same order as the multiplexer control, register enable, and carry input lines from the controller to the data-path. Additional columns are required for each cond enable and for signals that are generated by the controller. A signal generated by the controller has the form (signal signal-name) in the program specification. One design would have the cond enable columns on the left of the ROM, the signals generated by the controller in the center of the ROM, and the multiplexer control, register enable, and carry input lines for the data-path on the right. Using this design, the 23 columns of the ROM from left to right would be:

```
cond enable 1
cond enable 2
cond enable 3
cond enable 4
cond enable 5
charge-time
(mpx 1 2)
(control-line 1 1)
(mpx 2 2)
(mpx 2 1)
(mpx 4 2)
(mpx 4 1)
(control-line 4 1)
(mpx 7 2)
(control-line 7 1)
(mpx 8 2)
(mpx 8 1)
```

```
(mpx 11 3)
(mpx 11 2)
(mpx 11 1)
(control-line 11 1)
(mpx 12 2)
(mpx 12 1)
```

The rows of the ROM correspond to the states of the processes and the substates of the conds. This produces the following rows for the ROM from top to bottom:

```
process time-clock state off
process time-clock state on
process fare-clock state for-hire
process fare-clock state hired
cond 1 substate time-on
cond 1 substate t
cond 2 substate time-on
cond 2 substate t
cond 3 subsubstate (= timer maximum-time)
cond 3 subsubstate t
cond 4 substate hire
cond 4 substate t
cond 5 substate (not hire)
cond 5 substate (and charge-time mile-mark)
cond 5 substate charge-time
cond 5 substate mile-mark
cond 5 substate t
```

All of the multiplexer control, register enable, and carry input lines for each unit are associated with a single process. The time-clock process has control of the first, second, and fourth units, the charge-time signal, and the first three conds. The fare-clock process has control of the seventh, eighth, eleventh, and twelfth units and the last two conds. Each ROM column associated with signals controlled by a process must be set to a 1 or a 0 in every valid combination of states and substates of that process.

This information can be used to derive the contents of the microprogram ROM.

The top row of the ROM is assigned to the off state of the time-clock process. In this state, the first cond is enabled and the second and third conds are not enabled. The charge-time signal is 0. The (mpx 1 2) and (control-line 1 1) are 1 to load the time clock state register from the internal port (second) unit. If a hyphen is used to represent a null entry in the microprogram ROM, the top row of the ROM is:

```
1 0 0 - - 0 1 1 - - - - - - - - - - - - - - -
```

The second row of the ROM is assigned to the on state of the time-clock process. In this state, the first cond is not enabled and the second cond is enabled. The third cond is controlled by the second cond. The charge-time signal is controlled by the third cond. The (mpx 1 2) and (control-line 1 1) are 1 to load the time clock state register from the internal port (second) unit. The second row of the ROM is:

```
0 1 - - - - 1 1 - - - - - - - - - - - - - - -
```

The third row of the ROM is assigned to the for-hire state of the fare-clock process. In this state, the fourth cond is enabled and the fifth cond is not enabled. The (mpx 7 2) and (control-line 7 1) are 1 to load the fare clock state register (seventh unit) from the internal port (eighth) unit. The third row of the ROM is:

- - - 1 0 - - - - - - - - 1 1 - - - - - - - -

The fourth row of the ROM is assigned to the hired state of the fare-clock process. In this state, the fifth cond is enabled and the fourth cond is not enabled. The (mpx 7 2) and (control-line 7 1) are 1 to load the fare clock state register (seventh unit) from the internal port (eighth) unit. The fourth row of the ROM is:

- - - 0 1 - - - - - - - - 1 1 - - - - - - - -

The fifth row of the ROM is assigned to the time-on substate of the first cond. In this substate, the next state internal port (second unit) is set to generate a 1 for the on state so (mpx 2 1) is 1 and (mpx 2 2) is 0. The timer register (fourth unit) is set to 0 so (mpx 4 1) is 1, (mpx 4 2) is 0 and (control-line 4 1) is 1. The fifth row of the ROM is:

- - - - - - - - 0 1 0 1 1 - - - - - - - - - -

The sixth row of the ROM is assigned to the t substate of the first cond. In this substate, the next state internal port (second unit) is set to generate a 0 for the off state so (mpx 2 1) is 0 and (mpx 2 2) is 1. The timer register (fourth unit) does not change so (control-line 4 1) is 0. The sixth row of the ROM is:

- - - - - - - - 1 0 - - 0 - - - - - - - - - -

The seventh row of the ROM is assigned to the time-on substate of the second cond. In this substate, the third cond is enabled. The next state internal port (second unit) is set to generate a 1 for the on state so (mpx 2 1) is 1 and (mpx 2 2) is 0. The timer register (fourth unit) is loaded in all subsubstates of this substate so (control-line 4 1) is 1. The seventh row of the ROM is:

- - 1 - - - - - - 0 1 - - 1 - - - - - - - - - - -

The eighth row of the ROM is assigned to the t sub-state of the second cond. In this substate, the third cond is not enabled. The next state internal port (second unit) is set to generate a 1 for the on state so (mpx 2 1) is 1 and (mpx 2 2) is 0. The timer register (fourth unit) is loaded to 0 so (mpx 4 1) is 1, (mpx 4 2) is 0, and (control-line 4 1) is 1. The eighth row of the ROM is:

- - 0 - - - - - - 0 1 0 1 1 - - - - - - - - - - -

The same process can be used for the remainder of the rows. The full microprogram ROM is shown in Figure 7.1.

4. Control.1 Modifications

The functions in the control.1 program must be modified to produce a microprogram ROM from the new object file control element structure described in Section VII.E.1

5. Single Bit Data-Path

The current nMOS MSC performs all single bit processing in the Weinberger control logic array. The library program contains macros that convert all single bit

cond enable 1	cond enable 2	cond enable 3	cond enable 4	cond enable 5	charge-time	(mpx 1 2)	(control-line 1 1)	(mpx 2 2)	(mpx 2 1)	(mpx 4 2)	(mpx 4 1)	(control-line 4 1)	(mpx 7 2)	(control-line 7 1)	(mpx 8 2)	(mpx 8 1)	(mpx 11 3)	(mpx 11 2)	(mpx 11 1)	(control-line 11 1)	(mpx 12 2)	(mpx 12 1)	
1	0	0	-	-	0	1	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	time-clock state off
0	1	-	-	-	-	1	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	time-clock state on
-	-	-	1	0	-	-	-	-	-	-	-	-	1	1	-	-	-	-	-	-	-	-	fare-clock state for-hire
-	-	-	0	1	-	-	-	-	-	-	-	-	1	1	-	-	-	-	-	-	-	-	fare-clock state hired
-	-	-	-	-	-	-	-	0	1	0	1	1	-	-	-	-	-	-	-	-	-	-	cond 1 time-on
-	-	-	-	-	-	-	-	1	0	-	-	0	-	-	-	-	-	-	-	-	-	-	cond 1 t
-	-	1	-	-	-	-	-	0	1	-	-	1	-	-	-	-	-	-	-	-	-	-	cond 2 time-on
-	-	0	-	-	-	-	-	0	1	0	1	1	-	-	-	-	-	-	-	-	-	-	cond 2 t
-	-	-	-	-	1	-	-	-	-	0	1	-	-	-	-	-	-	-	-	-	-	-	cond 3 (= timer maximum-time)
-	-	-	-	-	0	-	-	-	-	1	0	-	-	-	-	-	-	-	-	-	-	-	cond 3 t
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	1	0	0	1	1	-	-	cond 4 hire
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	0	-	-	-	0	-	-	cond 4 t
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	0	-	-	-	0	-	-	cond 5 (not hire)
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	1	0	1	0	1	0	1	cond 5 (and charge-time mile-mark)
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	1	1	0	0	1	1	0	cond 5 charge-time
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	1	1	0	0	1	0	1	cond 5 mile-mark
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	1	-	-	-	0	-	-	cond 5 t

Figure 7.1 Taxi Microprogram ROM Design

functions into combinations of NOR functions that are specified in the object file control element. These functions cannot be accomplished by a microprogrammed controller. A set of cells for single bit functions are required and an area for single bit processing must be added to the MSC floor plan.

F. IMPLEMENTATION STATUS

The cells specified in Chapter VI are laid out and tested. A microprogrammed controller constructed from these cells and using sequencers constructed from data path units will be able to fulfill all MSC controller requirements stated in Section IV.B. Addition of the cells to MSC and extension of MSC for the new cells are not completed in this thesis and will require further research.

VIII. CONTROLLER TESTING

After completion of the microprogrammed controller implementation described in Chapter VII, the resulting silicon compiler will require extensive testing. A series of benchmark design specifications will have to be generated that test all types of controller configurations. These benchmarks should include designs that have multiple processes, nested cond statements, processes with many states, controller generated signals, counter-stack sequencers, counter-no-stack sequencers, and no-counter-no-stack sequencers. Each of the benchmarks should be exercised using the MSC internal design interpreter and the simulated outputs produced by test inputs should be recorded. Each benchmark design should then be output in Magic cif format. The cif files should be loaded into Magic and verified using the Magic internal design rule checker. Simulation files should be prepared by extracting the designs using the Magic extract command and converting the extract files to simulation files with the ext2sim program. The circuits should be checked with the CRYSTAL static electrical check command. Each simulation file should be tested using the ESIM event driver switch level simulator with the same test inputs as were used with the MSC internal design interpreter tests.

The ESIM simulated outputs should be the same as the MSC internal design interpreter outputs.

IX. CONCLUSIONS

A. CONTROLLER DESIGN

A microprogrammed controller for the MSC silicon compiler has been designed and tested. The remaining steps to fully implement the controller have also been described. The design of the controller followed a complete development process. Requirements for the controller were defined, alternative technologies and organizations were analyzed, a technology and organization were selected, and a design was produced using the selected technology and organization. Use of a well defined formal development process was very beneficial to this thesis research.

B. MSC STRUCTURE

The current internal structure of the MSC programs is very technology dependent. A major reorganization would improve the addition of other technologies to MSC. The library and organelles.l programs are not permanent parts of the MSC compiler. They are loaded at run time. There should be library and organelles.l programs that are unique for each technology. All of the technology dependent functions of MSC should be moved to these programs. The name of the technology should be added to the beginning of each program's name or they should be stored in different directories so that the only technology dependent functions loaded into MSC at run time are those for the selected technology.

C. ADDITIONAL RESEARCH

There are several topics for follow-on research that are presented in this thesis. The most important is the development of the cifsymbol macro described in Section VII.D. This would provide a technology independent method for adding cells to MSC. Other follow on research involves the implementation and test of the microprogram controller described in Section VII.E and Chapter VIII.

APPENDIX A ISI INSTALLATION LOG

This file is a log of the procedures used to compile the MacPitts Silicon Compiler on the ISI system at NPGS.

25 Feb 1987	J. Harmon and R. Limes loaded original source files from the computer science VAX installation into the /usr/macpit directory.
25 Feb 1987	J. Harmon modified the c-routines.c program to eliminate the multiple definition problem for the global variable "ospeed". This problem was the result of loading a C language object file into Franz Lisp or Liszt with a global variable which had the same name as one inside Lisp or Liszt. MacPitts was written using an older Opus of Franz that did not have a termcap interface. The current Franz has termcap capability that includes the global variable "ospeed" from the file tputs.o in libtermcap.a. C-routines.c uses the curses package that includes a global variable "ospeed" in cr_tty.o in libcurses.a. The fix was to include a local definition of the functions in cr_tty.o in libcurses.a. This fix replaces the previous fix of changing the lib files that returned the problem when Franz was recompiled or a new version of the lib files was installed.
26 Feb 1987	J. Harmon modified lincoln.l to remove compilation warnings that were caused by the upgrade to the current opus of Franz Lisp and Liszt. Also corrected errors that caused compilation warnings in the original opus. Escaped all $ characters so there would not be any trouble if the back-quote symbol is changed to $. Changed remove to remove-parameter since remove is a standard function in the current opus of Lisp. Removed the path restriction on the cfasl of c-routines.o so that a local version may be used. Declared fixnum, list, predicate, thing, and grade-predicate special for the whole program so that they are special when the macros that use them are evaluated. Removed the pp-form function since it used a Lisp primitive $prpr that is not in the current opus. The current opus has its own pp-form. Removed profane language from the err notice in tr-traceexit.

27 Feb 1987	J. Harmon added all of the changes written by E. Malagon for L5. This included adding messages to L5-err, adding SCMOS layers, creating a hybrid nMOS/SCMOS technology, and changing the CIF output from to use "94" extension for labels.
27 Feb 1987	J. Harmon compiled defstructs.l with no errors. No modification required for defstructs.l.
27 Feb 1987	J. Harmon escaped all $ characters in front-page.l.
27 Feb 1987	J. Harmon compiled general.l with no errors. No modification required for general.l.
27 Feb 1987	J. Harmon escaped all $ characters in prepass.l and changed (argv) to (argv -1) in those cases where it is to return the number of command line arguments. This is to conform with the argv function in the current opus of Franz Lisp.
27 Feb 1987	J. Harmon compiled extract.l with no errors. No modification required for extract.l
27 Feb 1987	J. Harmon compiled frame.l with no errors. No modification required for frame.l.
27 Feb 1987	J. Harmon compiled data-path.l with no errors. No modification required for data-path.l.
27 Feb 1987	J. Harmon compiled control.l with no errors. No modification required for control.l.
27 Feb 1987	J. Harmon compiled flags.l with no errors. No modification required for flags.l.
27 Feb 1987	J. Harmon compiled padgen.l with no errors. No modification required for padgen.l.
27 Feb 1987	J. Harmon ran padgen.l under Lisp to generate pads.l. Pads.l does not compare with the distribution pads.l. I will use the version generated in this installation and not the original version that may not be completely compatible with the ISI installation.
27 Feb 1987	J. Harmon compiled pads.l with no errors. The pads.l and pads.o files take up a lot of space! Disk is up to 99% full.

2 Mar 1987	The disk was cleaned up so that there now is lots of disk space.
2 Mar 1987	J. Harmon compiled order.l with no errors. No modification required for order.l.
2 Mar 1987	J. Harmon compiled general.l with no errors. No modification required for general.l.
2 Mar 1987	J. Harmon changed the name of the function defined in an .env file from environment to env. The environment function is a built-in function in the current opus of Franz Lisp. The env function in the .env file is used to simulate the environment outside the chip. If there is any outside processing of chip outputs to produce chip inputs, this may be automatically done by the env function. This processing is enabled/disabled by the e command to the interpreter. The env function is defined by a defun in the .env file and has an argument list of (clocked? new-states word-length definitions) where clocked? is a flag that is 't if the update is a clock cycle update and 'f if the update is a propagate update. New-states are the current states for the simulator, word-length is the data path word size, and definitions is a list of design definitions extracted from the design specification. The function env returns a list of new-states. The port-state-value or signal-state-value of the new-state is the value produced by the environment for that input. The port-state-driver or signal-state-driver must be set to 'environment. If a new-state is not returned that had been previously returned by a call to env, the system will return control of the signal to the chip. An env may not update a signal or port that is stable and set by the chip. If the port-state-driver or signal-state-driver for the old state is 'chip then the port-state-value or signal-state-value must be 'unset for the env to update the input port or signal.
2 Mar 1987	J. Harmon compiled organelles.l with no errors. No modification required for organelles.l.
6 Mar 1987	J. Harmon replaced all # symbols with the word "number" and appropriate hyphens in control.l, data-path.l, defstructs.l, extract.l, frame.l, front-page.l, general.l, interpret.l,

	lincoln.l, order.l, organelles.l, and Library. The # symbol is used in the reader as the vsplicing-macro character. The old version of MacPitts removed the special use of this symbol. The debugger and stepper do use this symbol so this violation of standard syntax interfered with the use of the debugger and stepper.
10 Mar 1987	J. Harmon renamed function number-of-tracks in control.l to find-number-of-tracks to avoid confusion with new variable named number-tracks. Note: there is a local variable used many times in organelles.l that is called number-of-tracks and a function called determine-number-of-tracks. These two functions in control.l and organelles.l should probably be combined and placed in general.l.
10 Mar 1987	J. Harmon replaced the name track-number in organelles.l with the name organelles-track-number to prevent conflict with variables now named track-number in other sections of MacPitts.
10 Mar 1987	J. Harmon replaced alpha form with mapcar form in Library for macro and, macro nand, and macro xor. Alpha expansion fo the old form did not work.
10 Mar 1987	J. Harmon compiled taxi program with no errors.
11 Mar 1987	J. Harmon updated Makefile for installation on ISI cluster workstations. Updated the install routine to copy the new executable MacPitts with Library and organelles.o to ISI1 and ISI2.
13 Mar 1987	J. Harmon updated all sources with comments IAW Brooks Programming In Common Lisp. Added make xref to the Makefile to automatically make and print a cross reference. Added a Logfile to the compilation steps in the Makefile to save the compilation messages in one log. Updated make clean in the Makefile to remove all unused files from the /usr/macpit directory. Compiled all sources without errors or warnings. Installed new version. Generated cross reference. Cleaned up directory. Made all source files read only. This established the initial baseline of MacPitts on the ISI.

APPENDIX B SOFTWARE CHANGE PROPOSAL SCP-2

SOFTWARE CHANGE/SOFTWARE ENHANCEMENT PROPOSAL
Based on DOD-STD-1679A(NAVY) DI-E-2117A

1. System/Project Name: MACPITTS

2. Date Prepared: 22 April 1987

3. SCP Number: 2

4. Title of SCP/SEP: Selectable Technology CIF Output

5. Originator: J. Harmon

6. Component(s) affected: L5.1 frame.1 prepass.1 Makefile

7. Description of Problem/Need for SCP/SEP:

   The current CIF output of MacPitts for 4 or 5 micron hybrid nMOS and SCMOS is not the same as the CIF used by the Magic and cifp programs. The only program that is compatible with the current CIF is the cifplot program from the Berkeley CAD tools. A MacPitts CIF layout is grossly distorted when loaded into Magic. The names of sub-modules in the MacPitts CIF are not recognized by Magic and input/output signal labels are not attached to the proper layers. Any designs with both nMOS and SCMOS layers is not acceptable to Magic. A CIF file must be able to be loaded into Magic to be extracted for the Crystal, ESIM, and RSIM programs. The cifp program used to plot CIF on the color plotter connected to the ISI workstations does not accept the format of the MacPitts CIF.

8. Description of Recommended SCP/SEP:

   This change adds new command line options to MacPitts named "magic", "nmos", "cmos", "scmos", "hybrid", "3u", "2u", and "1.2u". If "magic" is included in the command line, MacPitts will produce a CIF file that is compatible with Magic and cifp. The "nmos", "cmos", and "scmos" command line options specify that the CIF output contains only the layers that are valid for that technology. The "hybrid" command line option specified that the CIF output contains both nMOS and SCMOS layers. This CIF cannot be loaded into Magic but can be plotted with the cifplot and cifp programs. The "3u", "2u", and "1.2u" command line options specify minimum feature sizes of 3, 2, and 1.2 microns, respectively. Each technology has a default minimum feature size and a list of valid minimum feature sizes. The program will not accept an invalid combination of minimum feature size and technology. A new technology or minimum feature size entry on the command line replaces any previous entries and defaults. The CIF file includes names of sub-modules that are used by Magic and input/output labels that are connected to the proper layers. The default technology is nMOS and the default CIF format is compatible with Magic. Each technology has a default minimum feature size.

9. Alternatives/Impact if not Approved:

A conversion program may be written to convert from MacPitts CIF to Magic CIF and filter undesired layers. The program may also be required to modify the minimum feature size. If MacPitts CIF cannot be loaded properly into Magic, then the Magic design rule checker and the Crystal, ESIM, and RSIM simulators cannot be used on MacPitts layouts.

10. Baselines Affected: 13 March 1987 Version 1.00 MacPitts

11. Documentation/Specifications Affected:

RVLSI-3, An Introduction to Macpitts, J.R. Southard 10 February 1983

RVLSI-5, L5 User's Guide, K.W. Crouch 7 March 1984

12. Other Systems or Configuration Items Affected: None.

13. Effect of SCP/SEP on System Employment, Integrated Logistics Support (ILS), Training, Effectiveness, etc:

This SCP would increase the number of programs that could use MacPitts CIF. This would increase the employment and effectiveness of MacPitts. There would be no change in ILS. Use of the magic command line option of MacPitts and procedures for using MacPitts layouts in Magic, Crystal, ESIM, and RSIM would have to be included in the Silicon Compiler Training Syllabus.

14. Net Effect on System Resources (e.g., Processing Time, Memory/Disk Space):

This change adds less than 500 lines of changes and additions to the MacPitts source code, including comments and separator lines. The size of the compiled program is not significantly increased. Run time memory requirements on a sample design should increase by 10% and design compile time should increase by 10% to 20%.

15. Developmental Requirements: 4 man-weeks for recoding and testing.

16. SCP/SEP Effectivity Point: As soon as possible

17. Date Approval Needed By: No applicable established schedule.

18. This SCP/SEP Must be Accomplished Before/With/After:

No relationship to other pending modifications.

19. Supersedes or Replaces SCP/SEP/STR:

This SCP supersedes SCP-1/

20. Approved By:

21. Approval Date:

22. Changes Installed By:

23. Installation Date:

Testing Results:

Several designs, including incrementer.mac and taxi.mac, have been compiled with these changes and successfully loaded into Magic and cifp. The design are not distorted, the sub-module names are recognized by Magic, and the input/output labels are connected to the proper layers.

A run without the magic option selected produced the same CIF output as the baseline program without modifications with the exception of the system identification comment added in the modifications.

ROM0, ROM1, and ROMnull cells with SCMOS layers have been generated and loaded into Magic.

All changes listed in the remainder of this software change
     proposal are a combination of modified original
     MacPitts programs and new programs.

The following are changes to L5.1:

```
;;;;;; >>>>>> Add the following to the modification history:

;;;  8 Apr 1987 J. Harmon added the capability to output CIF
;;;             in Magic compatible format by adding a new
;;;             global variable, --L5-magic-flag, an
;;;             initialization procedure for the new
;;;             variable, and new functions magic! and
;;;             magic? to change and query the value of the
;;;             variable respectively.  Added conditional
;;;             processing to the cifout, cifout-external-
;;;             names, cifout-external-name, cifout-define-
;;;             symbol, cifout-call-symbol, move-symbol,
;;;             cifout-rect, and merge-cifout-files
;;;             functions.  Upgraded the lambda-to-
;;;             centimicron function to convert lambda
;;;             measurements to scaled Magic measurements.
;;;
;;; 14 Apr 1987 J Harmon added new global variables --L5-
;;;             scale-factor-a and --L5-scale-factor-b to L5
;;;             for scaling the numbers in the cifout
;;;             section.  The number output is the number of
;;;             centimicrons times --L5-scale-factor-b and
;;;             divided by --L5-scale-factor-a.  Added new
;;;             initialization for the two new global
;;;             variables.  Updated lambda-to-centimicron to
;;;             use the new scale factors.  Added new
;;;             function set-scale-factors to set the values
;;;             of the scale factors and scale-factor-a? and
;;;             scale-factor-b? to return the values of the
;;;             scale factors.  Updated cifout-define symbol
;;;             to use the new scale factors.  Updated
;;;             allowed-layers to remove SCMOS layers from
;;;             the nMOS technology and add the SCMOS, CMOS,
;;;             and hybrid technologies.  Added allowed-
;;;             technologies? to test for a valid
;;;             technology, default-minimum-feature-size and
;;;             allowed-feature-sizes to return the default
;;;             and valid minimum feature sizes for the
;;;             current technology, and pad-glass-layer? to
;;;             return the layer for the pad holes in the
;;;             over-glass layer in the current technology.
;;;             Added convert-feature-size to convert a
;;;             minimum feature size command line option to
;;;             a lambda spacing.  updated rect, box, and
;;;             mark to process layers correctly.
```

```
;;;
;;; 16 Apr 1987 J Harmon changed the name of minimum-
;;;             feature-size! and minimum-feature-size
;;;             functions tc lambda-spacing! and lambda-
;;;             spacing ana changed the name of the global
;;;             variable --L5-minimum-feature-size to --L5-
;;;             lambda-spacing to avoid confusion.  The
;;;             value used in these functions and variable
;;;             is for the lambda spacing and not the
;;;             minimum feature size that is twice the
;;;             lambda spacing.
;;;
;;; 16 Apr 1987 J Harmon added defsymbols for common SCMOS
;;;             contacts.
;;;

;;;;;>>>> Replace the following declaration in Headers
section

(declare
  ;;  8 Apr 87  J Harmon added new global variable --L5-
  ;;            magic-flag to be used to select magic style
  ;;            cif output.  A value of nil or () disables
  ;;            magic cif output and a value of t or non-nil
  ;;            value enables magic cif output.
  ;;
  ;; 14 Apr 87  J Harmon added new global variables --L5-
  ;;            scale-factor-a and --L5-scale-factor-b to
  ;;            specify the scaling used in CIF output.
  ;;
  ;; 16 Apr 87  J Harmon changed name of --L5-minimum-
  ;;            feature-size to --L5-lambda-spacing for
  ;;            clarity.  The minimum feature size is two
  ;;            times the lambda spacing.
  (special --L5-magic-flag
           --L5-scale-factor-a
           --L5-scale-factor-b
           --L5-symbol-port
           --L5-symbol-file
           --L5-symbol-list
           --L5-symbol-number
           --L5-technology
           ;;jh changed name of --L5-minimum-feature-size to
           ;;   --L5-lambda-spacing for clarity.  The
           ;;   minimum feature size is two times the lambda
           ;;   spacing.
           --L5-lambda-spacing
           attributes
           dx
           dy
           name
```

```
              item-tree
              path
              symbol
              symbol-id))

;;;;;>>>Add or replace the following setqs to the Global
;;;;;>>>Variables part of the Definitions section.

(setq --L5-magic-flag ())
 ;;  8 Apr 87 J Harmon added initialization for the new --
 ;;           L5-magic-flag variable.  The initial setting
 ;;           is with magic cif output disabled.

(setq --L5-scale-factor-a 1)
 ;; 14 Apr 87 J Harmon added initialization for the new
 ;;           --L5-scale-factor-a variable.  The initial
 ;;           setting is 1.

(setq --L5-scale-factor-b 1)
 ;; 14 Apr 87 J Harmon added initialization for the new
 ;;           --L5-scale-factor-b variable.  The initial
 ;;           setting is 1.

(setq --L5-lambda-spacing 200)
 ;; 16 Apr 87 J Harmon changed name of --L5-minimum-feature-
 ;;           size to --L5-lambda-spacing for clarity.  The
 ;;           minimum feature size is two times the lambda
 ;;           spacing.  Also made the default lambda spacing
 ;;           200 centimicrons.

;;;;;>>>Add or replace the following functions to the
;;;;;>>>Changeable common definitions part of the Common
;;;;;>>>functions section

(def lambda spacing!
 ;; 16 Apr 87 J Harmon renamed minimum-feature-size!
 ;;             function to lambda-spacing! and replaced all
 ;;             references to minimum feature size to lambda
 ;;             spacing to avoid confusion.  The actual
 ;;             value used in the program is the lambda
 ;;             spacing and not the minimum feature size
 ;;             that is two times the lambda spacing.
 (lambda (lambda-spacing)
  (setq --L5-lambda-spacing lambda-spacing)))

(def lambda spacing
 ;; 16 Apr 87 J Harmon renamed minimum-feature-size
 ;;           function to lambda-spacing and replaced all
 ;;           references to minimum feature size to lambda
 ;;           spacing to avoid confusion.  The actual
 ;;           value used in the program is the lambda
 ;;            spacing and not the minimum feature size
 ;;            that is two times the lambda spacing.
 (lambda ()
  --L5-lambda-spacing))

(def magic!
 ;;  8 Apr 87 J Harmon added the magic! function that sets
 ;;            the value of the L5 global variable --L5-
 ;;            magic-flag to the argument of the magic!
 ;;            function.  This allows separately compiled
 ;;            procedures to set the flag without direct
 ;;            access to the variable.
 (lambda (magic-flag)
  (setq --L5-magic-flag magic-flag)))

(def magic?
 ;;  8 Apr 87 J Harmon added the magic? function that reads
 ;;            the value of the L5 global variable --L5-
 ;;            magic-flag and returns it as the value of the
 ;;            magic? function.  This allows separately
 ;;            compiled procedures to read the flag without
 ;;            direct access to the variable.
 (lambda ()
  --L5-magic-flag))

(def set-scale-factors
 ;; 14 Apr 87 J Harmon added set-scale-factors function that
 ;;            sets the values of the two new variables --L5-
 ;;            scale-factor-a and --L5-scale-factor-b to the
 ;;            values passed to the function.
 (lambda (scale-a scale-b)
  (setq --L5-scale-factor-a scale-a)
  (setq --L5-scale-factor-b scale-b)))

(def scale-factor-a
 ;; 14 Apr 87 J Harmon added a new scale-factor-a function
 ;;            that returns the value of the new variable --
 ;;            L5-scale-factor-a
 (lambda ()
  --L5-scale-factor-a))

(def scale-factor-b
 ;; 14 Apr 87 J Harmon added a new scale-factor-b function
 ;;            that returns the value of the new variable --
 ;;            L5-scale-factor-b
 (lambda ()
  --L5-scale-factor-b))

(def allowed-technologies
 ;; 14 Apr 87 J Harmon added allowed-technologies function
 ;;            to L5.  This function returns a list of
 ;;            allowed technologies.
 (lambda ()
  '(nmos cmos scmos hybrid)))

(def allowed-layers
 ;; 23 Jan 87 E. Malagon added allowed layers.  Allows
 ;;           changing from default nmos to cmos, scmos.
 ;;           Adds hybrid nmos technology that has both nmos
 ;;           and scmos layers.
 ;; 14 Apr 87 J Harmon added hybrid technology and reduced
 ;;           nmos layers to actual nmos layers.
 (lambda ()
  (cond
   ((eq 'nmos (technology)) '(ND NP NM NI NC NG NB NX XP))
   ((eq 'hybrid (technology)) '(ND NP NM NI NC NG NB NX XP
         CPG CAA CMF CMS CSN CSP CCP CCA CVA COG CWP CWN))
   ((eq 'cmos (technology)) '(CD CP CM CM2 CS CC CG CW NX
         XP))
   ((eq 'scmos (technology)) '(CPG CAA CMF CMS CSN CSP CCP
         CCP CVA COG CWP CWN))
   (t (L5-err
            '|That technology is not recognized by L5|)))))

(def default-minimum-feature-size
 ;; 14 Apr 87 J Harmon added new function default-minimum-
 ;;           feature-size that returns a default minimum
 ;;           feature size for the current L5 technology.
 (lambda ()
  (cond
   ((eq 'nmos (technology)) '4u)
   ((eq 'hybrid (technology)) '3u)
   ((eq 'cmos (technology)) '3u)
   ((eq 'scmos (technology)) '3u)
   (t (L5-err
            '|That technology is not recognized by L5|)))))

(def allowed-minimum-feature-sizes
 ;; 14 Apr 87 J Harmon added a new function allowed-minimum-
 ;;           feature-sizes that returns a list of the
 ;;           allowed minimum feature sizes for the current
 ;;           technology.
 (lambda ()
  (cond
   ((eq 'nmos (technology)) '(4u 5u))
   ((eq 'hybrid (technology)) '(1.2u 2u 3u 4u 5u))
   ((eq 'cmos (technology)) '(4u 5u))
   ((eq 'scmos (technology)) '(1.2u 2u 3u))
   (t (L5-err
   '|That minimum feature size is not recognized by L5|)))))
```

```
(def convert-feature-size
 ;; 14 Apr 87 J Harmon added a new function convert-feature-
 ;;            size that accepts a minimum-feature-size in
 ;;            option-list format and returns the lambda
 ;;            spacing in centimicrons.
 (lambda (min-size)
  (cond
   ((eq '5u min-size) 250)
   ((eq '4u min-size) 200)
   ((eq '3u min-size) 150)
   ((eq '2u min-size) 100)
   ((eq '1.2u min-size) 60)
   (t (L5-err '|That minimum feature size is not recognized
by L5|)))))

(def pad-glass-layer
 ;; 14 Apr 87 J Harmon added function pad-glass-layer that
 ;;            returns the glass layer associated with pads
 ;;            in the current technology.
 (lambda ()
  (cond
   ((eq 'nmos (technology)) 'NG)
   ((eq 'hybrid (technology)) 'NG)
   ((eq 'cmos (technology)) 'CG)
   ((eq 'scmos (technology)) 'COG)
   (t (L5-err
            '|That technology is not recognized by L5|)))))

;;;>>> Replace the following in the Item Creation section:

(def rect
 ;; 14 Apr 87 J Harmon removed the test for a valid layer
 ;;            from the rect function.  This allows the
 ;;            cifout-rect function to determine if a
 ;;            rectangle should be put into the output.
 (lambda (layer Xmin Ymin Xmax Ymax)
  (cond
   ;; jh removed test for valid layer from rect function.
   ;;    This allows the cifout-rect function to determine
   ;;    if a rectangle should be put into the output.
   ((greaterp Xmin Xmax) (L5-err
            '|Xmin should be less than or equal to Xmax|))
   ((greaterp Ymin Ymax) (L5-err
            '|Ymin should be less than or equal to Ymax|))
   (t
    (make-item
     Xmin
     Ymin
     Xmax
     Ymax
     ()
     ()
     (make-rect-tree layer Xmin Ymin Xmax Ymax ))))))

(def box
 ;; 14 Apr 87 J Harmon removed the test for a valid layer
 ;;           from the box function.  This allows the
 ;;           cifout-rect function to determine if a
 ;;           rectangle should be put into the output.
 (lambda (layer length width xcenter ycenter)
  (cond
   ;; jh removed test for valid layer from box function.
   ;;    This allows the cifout-rect function to determine
   ;;    if a rectangle should be put into the output.
   ((lessp length 0) (L5-err '|Illegal box length|))
   ((lessp width 0) (L5-err '|Illegal box width|))
   (t
    (let ((Xmin (diff xcenter (quotient length 2.0)))
          (Ymin (diff ycenter (quotient width 2.0)))
          (Xmax (plus xcenter (quotient length 2.0)))
          (Ymax (plus ycenter (quotient width 2.0))))
     (make-item
      Xmin
      Ymin
      Xmax
      Ymax
      ()
      ()
      (make-rect-tree layer Xmin Ymin Xmax Ymax)))))))

(def mark
 ;; 14 Apr 87 J Harmon removed the test for a valid layer
 ;;           from the mark function.  This allows the
 ;;           cifout-rect function to determine if a
 ;;           rectangle should be put into the output.
 (lambda (name x y layer attributes)
  (cond
   ((not (number? x))
    (L5-err '|Point x coordinate must be a number|))
   ((not (number? y))
    (L5-err '|Point y coordinate must be a number|))
   ;; jh removed test for valid layer from mark function.
   ;;    This allows the cifout-rect function to determine
   ;;    if a rectangle should be put into the output.
   ((not (list? attributes))
    (L5-err '|Point attributes must be in a list|))
   (t
    (make-item
     x
     y
     x
     y
     (make-points (make-point (list name) x y layer
                              attributes))
     ()
     (make-null-tree))))))

;;;;;>>>>>Replace the following functions in the Item Cifout
;;;;;>>>>>section

(def cifout
 ;;  8 Apr 87 J Harmon added Magic compatible cifout
 ;;          operations.
 ;; 16 Apr 87 J Harmon replaced the function minimum-
 ;;          feature-size with the new function name
 ;;          lambda-spacing in cifout to avoid confusion.
 ;;          the value used is the lambda spacing and not
 ;;          the minimum feature size that is twice the
 ;;          lambda spacing.
 (lambda (item file title)
  (cond
   ((null item) (L5-err '|Cannot cifout null item|))
   (t (let ((cifout-file (concat file '.cif))
            (cifout-port1
             (outfile (concat (scratch-directory) '/ file
                              '.cif 1)))
            (cifout-file1 (concat (scratch-directory) '/
                                  file '.cif 1))
            (cifout-port2
             (outfile (concat (scratch-directory) '/ file
                              '.cif 2)))
            (cifout-file2 (concat (scratch-directory) '/
                                  file '.cif 2)))
        (terpri (L5-symbol-port))
        (patom "(Title : " cifout-port1)
        (patom title cifout-port1)
        (patom ");" cifout-port1)
        (terpri cifout-port1)
        (patom "(U.S. Naval Postgraduate School);"
               cifout-port1)
        (terpri cifout-port1)
        ;; jh added system identification comment to the CIF
        ;;    header.
        (patom "(ECE Department ISI System);" cifout-port1)
        (terpri cifout-port1)
        (patom "(lambda is " cifout-port1)
        (patom (lambda-spacing) cifout-port1)
        (patom " centimicrons);" cifout-port1)
        (terpri cifout-port1)
        ;; jh added symbol 1 with a scaling factor of 50 to
        ;;    the CIF header.  Also added CIF User Extension
        ;;    9 with the title to make the name of the new
        ;;    symbol 1 be the title of the design.  Also
        ;;    added processing for the magic option.
        (cond ((magic?) (patom "DS 1 " cifout-port1)
                        ;; 14 Apr 87 jh added variable scale
                        ;;                factors
                        (patom (scale-factor-a) cifout-port1)
                        (patom " " cifout-port1)
                        (patom (scale-factor-b) cifout-port1)
                        (patom ";" cifout-port1)
                        (terpri cifout-port1)
                        (patom "9 " cifout-port1)
                        (patom title cifout-port1)
                        (patom ";" cifout-port1)
                        (terpri cifout-port1)))
        (cifout-external-names
             (find-attributes item '(external)) cifout-port1)
        ;; jh deleted extra line for magic CIF output
        (cond ((not(magic?)) (terpri cifout-port1)))
        ;; jh added processing to determine which file gets
        ;;    the tree output
        (cifout-tree-walker (item-tree item) ()
                            (cond ((magic?) cifout-port1)
                                  (t cifout-port2)))
        ;; jh added end of symbol 1 for magic.
        (cond ((magic?) (patom "DF;" cifout-port1)
                        (terpri cifout-port1))
               (t (patom "End" cifout-port2)))
        ;; jh added processing to determine which file gets
        ;;    end of line.
        (terpri (cond ((magic?) cifout-port1)
                      (t cifout-port2)))
        ;; jh added call to symbol 1 and end for magic CIF
        ;;    output
        (cond ((magic?) (patom "C 1;" cifout-port2)
                        (terpri cifout-port2)
                        (patom "End" cifout-port2)
                        (terpri cifout-port2)))
        (merge-cifout-files cifout-file cifout-file1
                            cifout-file2)
        (close cifout-port1)
        (close cifout-port2))))))

(def cifout-external-names
 ;; 24 Jan 87 E. Malagon added the char 't' to the call to
 ;;           cifout-external-name in cifout-external-names
 ;;           to initiate non-MIT option.  This allows
 ;;           points to be plotted with their labels.  '94'
 ;;           extension is more conventional than the 'O'
 ;;           extension.
 ;;  8 Apr 87 J. Harmon removed extra line for Magic CIF
 (lambda (points cifout-port)
  (cond
    ;; jh removed extra line for magic CIF option
    ((null points) (cond ((not(magic?))
                                   (terpri cifout-port))))
    (t
     (cifout-external-name (car points) cifout-port t)
     (cifout-external-names (cdr points) cifout-port)))))

(def cifout-external-name
 ;; 24 Jan 87 E. Malagon added non-MIT option to cifout-
 ;;           external-name
 ;; 25 Mar 87 J. Harmon changed name of VDD to Vdd for
 ;;           proper simulator input
 ;; 25 Mar 87 J. Harmon added exclamation point to end of
 ;;           external names for proper simulator input.
 ;;  8 Apr 87 J. Harmon added rectangles of metal and pad
 ;;           for proper binding of labels to materials in
 ;;           Magic.
 ;; 14 Apr 87 J Harmon added processing to make cifout-
 ;;           external-name technology independent.  it only
 ;;           outputs an external name if it is for a valid
 ;;           layer in the current technology.  The layer
 ;;           for the hole in the over-glass is determined
 ;;           by the pad-glass-layer function.  The layer
 ;;           used is based on the current technology.
 (lambda (point cifout-port &optional non-MIT)
  (let ((name (capitalize (atomize-external-name
                                    (point-name point)))))
    ;; jh added a box of metal under each external name so
    ;;    that magic leaves the label for the name connected
    ;;    to metal rather than move it to space when the CIF
    ;;    file is loaded into magic.
    ;; jh added test for a valid layer
    (cond ((member (point-layer point) (allowed-layers))
           (cond ((magic?) (patom "L " cifout-port)
                  (patom (point-layer point) cifout-port)
                  (patom "; B L " cifout-port)
                  (patom (lambda-to-centimicron 4)
                                             cifout-port)
                  (patom " W " cifout-port)
                  (patom (lambda-to-centimicron 4)
                                             cifout-port)
                  (patom " C " cifout-port)
                  (patom (lambda-to-centimicron
                                (point-x point)) cifout-port)
                  (patom ", " cifout-port)
                  (patom (lambda-to-centimicron
                                (point-y point)) cifout-port)
                  (patom ";" cifout-port)
                  (terpri cifout-port)
                  (patom "L " cifout-port)
                  ;; jh replaced fixed glass layer with a
```

```
          ;;    call to the function that returns the
          ;;    glass layer for the current
          ;;    technology.
          (patom (pad-glass-layer) cifout-port)
          (patom "; B L " cifout-port)
          (patom (lambda-to-centimicron 4)
                                         cifout-port)
          (patom " W " cifout-port)
          (patom (lambda-to-centimicron 4)
                                         cifout-port)
          (patom " C " cifout-port)
          (patom (lambda-to-centimicron
                         (point-x point)) cifout-port)
          (patom ", " cifout-port)
          (patom (lambda-to-centimicron
                         (point-y point)) cifout-port)
          (patom ";" cifout-port)
          (terpri cifout-port)))
   (cond (non-MIT (patom "94 " cifout-port)
          ;; jh Changed name of VDD to Vdd for proper
          ;;    simulator input.
          (cond ((and (magic?) (equal name 'VDD))
                 (patom "Vdd" cifout-port)
                (t (patom name cifout-port)))
          ;; jh Added exclamation point to end of
          ;;    external names for proper simulator
          ;;    input.
          (cond ((magic?) (patom "!" cifout-port)))
          (patom " " cifout-port)
          (patom (lambda-to-centimicron
                         (point-x point)) cifout-port)
          (patom " " cifout-port)
          (patom (lambda-to-centimicron
                         (point-y point)) cifout-port)
          (patom " " cifout-port)
          (patom (point-layer point) cifout-port)
          (patom ";" cifout-port)
          (terpri cifout-port))
         (t (patom "0 " cifout-port)
          (patom (lambda-to-centimicron
                         (point-x point)) cifout-port)
          (patom " " cifout-port)
          (patom (lambda-to-centimicron
                         (point-y point)) cifout-port)
          (patom " " cifout-port)
          (patom (point-layer point) cifout-port)
          (patom " N " cifout-port)
          (patom name cifout-port)
          (patom ";" cifout-port)
          (terpri cifout-port)))))))
```

```
(def cifout-define-symbol
 ;;  8 Apr 87 J Harmon added Magic compatible define symbol
 ;;            output.
 ;; 14 Apr 87 J Harmon added scale factor output to cifout-
 ;;            define-symbol
 (lambda (item function-name cifout-port)
  (patom "DS " cifout-port)
  ;; jh Added 1 to the structure number for magic CIF
  ;;    output.  Structure number 1 is the whole design.
  ;;    added scale factor for magic.
  (cond ((magic?)
         (patom (1+ (length (L5-symbol-list))) cifout-port)
         (patom " " cifout-port)
         ;; jh added the scale factor printing from the new
         ;;    scale-factor-a and scale-factor-b functions.
         (patom (scale-factor-a) cifout-port)
         (patom " " cifout-port)
         (patom (scale-factor-b) cifout-port)
        (t (patom (length (L5-symbol-list)) cifout-port)))
  (patom ";" cifout-port)
  (terpri cifout-port)
  ;; jh Added function 9 for magic CIF output to name a
  ;;    structure with the function name.
  (cond ((magic?)
         (patom "9 " cifout-port)
         (patom function-name cifout-port)
         (patom ";" cifout-port)
        (t
         (patom "(name: " cifout-port)
         (patom function-name cifout-port)
         (patom ");" cifout-port)))
  (terpri cifout-port)
  (patom "(bounding box: " cifout-port)
  (patom (left item) cifout-port)
  (patom "," cifout-port)
  (patom (bottom item) cifout-port)
  (patom "," cifout-port)
  (patom (right item) cifout-port)
  (patom "," cifout-port)
  (patom (top item) cifout-port)
  (patom ");" cifout-port)
  (terpri cifout-port)
  (patom "DF;" cifout-port)
  (terpri cifout-port)))

(def cifout-call-symbol
 ;;  8 Apr 87 J Harmon added 1 to symbol numbers for Magic
 ;;            CIF output
 (lambda (symbol-call-tree stack cifout-port)
  (cond
   (t
    (patom "C " cifout-port)
    ;; jh added 1 to symbol numbers for magic CIF since
    ;;    symbol 1 is the whole design
    (cond ((magic?)
           (patom
            (1+ (symbol-call-tree-name symbol-call-tree))
            cifout-port))
          (t
           (patom
            (symbol-call-tree-name symbol-call-tree)
            cifout-port)))
    (patom " " cifout-port)
    (cifout-called-symbol-operation-handler stack
     cifout-port)
    (patom ";" cifout-port)
    (terpri cifout-port)))))

(def cifout-rect
 ;; 14 Apr 87 J Harmon added a test for valid layers to
 ;;           cifout-rect.
 (lambda (rect-tree cifout-port)
  (cond
   ((or (=0 (rect-width rect-tree))
        (=0 (rect-length rect-tree)))
    ())
   ;; jh added a test for valid layer to cifout-rect
   ((member (rect-tree-layer rect-tree) (allowed-layers))
    (let ((center (rect-center rect-tree)))
     (patom "L " cifout-port)
     (patom (rect-tree-layer rect-tree) cifout-port)
     (patom "; B L " cifout-port)
     (patom (lambda-to-centimicron (rect-length rect-tree))
            cifout-port)
     (patom " W " cifout-port)
     (patom (lambda-to-centimicron (rect-width rect-tree))
            cifout-port)
     (patom " C " cifout-port)
     (patom (lambda-to-centimicron (car center))
            cifout-port)
     (patom ", " cifout-port)
     (patom (lambda-to-centimicron (cadr center))
            cifout-port)
     (patom ";" cifout-port)
     (terpri cifout-port))))))

(def lambda-to-centimicron
 ;; 14 Apr 87 J Harmon added processing of scale factors in
 ;;           the function lambda-to-centimicron.
 ;; 16 Apr 87 J Harmon replaced minimum-feature-size with
 ;;           lambda-spacing in lambda-to-centimicron to
 ;;           avoid confusion.  The value used is the lambda
 ;;            spacing and not the minimum feature size that
 ;;            is twice the lambda spacing.
 (lambda (number)
  (fix (plus .5 (quotient
                 (times (scale-factor-b)
                        (times (lambda-spacing) number))
                        (scale-factor-a))))))

;;;;;>>>>>Add the following heading, note, and functions to
;;;;;>>>>>the Basic Items section:

;;; *** SCMOS Contacts ***

;;; Note:  The substrate contacts called nsubstratencontact,
;;;        nncontact, nsc, nnc, psubstratepcontact,
;;;        ppcontact, psc, and ppc are not included as
;;;        default structures since they do not act properly
;;;        in Magic as separate symbols.  They produce
;;;        inconsistencies between layers in different
;;;        cells.  These contacts must be built inside each
;;;        layout.

(defsymbol polycontact ()
 ;; 16 Apr 87 J Harmon added a defsymbol for a SCMOS
 ;;            polycontact.
 (merge
  (rect 'CPG 0 -4 4 0)
  (rect 'CMF 0 -4 4 0)
  (rect 'CCP 1 -3 3 -1)))

(def pcontact
 ;; 16 Apr 87 J Harmon added a function pcontact that
 ;;            returns a polycontact symbol
 (lambda ()
  (polycontact)))

(def pc
 ;; 16 Apr 87 J Harmon added a function pc that returns
 ;;            a polycontact symbol
 (lambda ()
  (polycontact)))

(defsymbol ndcontact ()
 ;; 16 Apr 87 J Harmon added a defsymbol for a SCMOS
 ;;            ndcontact.
 (merge
  (rect 'CAA 0 -4 4 0)
  (rect 'CMF 0 -4 4 0)
  (rect 'CCA 1 -3 3 -1)
  (rect 'CWP -5 -9 9 5)))
```

```
(def ndc
 ;; 16 Apr 87 J Harmon added a function ndc that returns
 ;;            a ndcontact symbol
 (lambda ()
  (ndcontact)))

(defsymbol pdcontact ()
 ;; 16 Apr 87 J Harmon added a defsymbol for a SCMOS
 ;;           pdcontact.
 (merge
  (rect 'CAA 0 -4 4 0)
  (rect 'CMF 0 -4 4 0)
  (rect 'CCA 1 -3 3 -1)
  (rect 'CSP -2 -6 6 2)))

(def pdc
 ;; 16 Apr 87 J Harmon added a function pdc that returns
 ;;            a pdcontact symbol
 (lambda ()
  (pdcontact)))

(defsymbol m2contact ()
 ;; 16 Apr 87 J Harmon added a defsymbol for a SCMOS
 ;;           m2contact.
 (merge
  (rect 'CMF 0 -4 4 0)
  (rect 'CMS 0 -4 4 0)
  (rect 'CVA 1 -3 3 -1)))

(def m2c
 ;; 16 Apr 87 J Harmon added a function m2c that returns
 ;;            a m2contact symbol
 (lambda ()
  (m2contact)))

(def via
 ;; 16 Apr 87 J Harmon added a function via that returns
 ;;            a m2contact symbol
 (lambda ()
  (m2contact)))

(def v
 ;; 16 Apr 87 J Harmon added a function v that returns
 ;;            a m2contact symbol
 (lambda ()
  (m2contact)))
```

Make the following changes to prepass.l:

```
;;;;;>>>>> Add the following to the modification history:

;;;  8 Apr 87 J Harmon added the capability to output CIF in
;;;           Magic compatible format by adding a new
;;;           command line option "magic".  Added processing
;;;           for the new option to the macpitts-compiler
;;;           and process-option functions.
;;; 16 Apr 87 J Harmon added processing to macpitts-compiler
;;;           to accommodate technologies specified on the
;;;           command line and add additional minimum
;;;           feature sizes.  Added new functions process-
;;;           command-line-options, process-technology-
;;;           option, process-minimum-feature-size, and
;;;           process-magic-option to support technology
;;;           selection.  Upgraded option processing to have
;;;           a new technology selection replace all current
;;;           technology selections and for a new minimum
;;;           feature size selection to replace all current
;;;           minimum feature size selections.  included a
;;;           default technology, and default minimum
;;;           feature sizes for each technology.  Included
;;;           checking for valid technology/minimum feature
;;;           size combinations.  Included processing to
;;;           establish scale factors for CIF output.

;;;;;>>> add or replace the following functions in
;;;;;>>> prepass.l:

(def macpitts-compiler
 ;; This function if the executive program for the macpitts
 ;; compiler.  It includes command line option processing
 ;; and calls all programs for loading the library file,
 ;; producing object file output, producing CIF output, and
 ;; for functionally simulating the design specification.
 ;;
 ;;  8 Apr 87 J Harmon added processing in macpitts-compiler
 ;;           to process the magic command line option.  The
 ;;           processing includes setting the new magic flag
 ;;           in L5 to true and setting the minimum-feature-
 ;;           size to 200.  This will be used with a
 ;;           reduction of 50 used in magic.  The reduction
 ;;           factor is the scaling in the DS lines for the
 ;;           CIF that has a=100 and b=2 meaning that the
 ;;           reduction factor = a/b is 50.  This is
 ;;           multiplied times all values in the CIF to get
 ;;           the actual value in centimicrons.
 ;; 16 Apr 87 J Harmon added processing for scmos, cmos, and
 ;;           hybrid technologies and different minimum
 ;;           feature sizes to macpitts-compiler.  Included
 ;;           calls to new functions to process-command-
 ;;           line-options, process-technology-option,
;;          process-minimum-feature-size, and process-
;;          magic-option.
(lambda (operands)
 (prog (file-name file object obj item)
  (setq initial-ptime (ptime))
  (setq initial-gccount \$gccount\$)
  (cond ((or (not (list? operands))
             (null operands)
             (not (atom? (car operands))))
         (patom "usage: (macpitts <filename> [<options>])")
         (terpr)
         (return ())))
  (setq file-name (car operands))
  ;; jh replaced old process-option function with new
  ;;    process-command-line-options function.  The new
  ;;    function updates the option list with the options
  ;;    specified on the command line.
  (process-command-line-options (cdr operands))
  (statistic (concat "for project " file-name))
  (statistic (concat "options: "
                     (slash (explode option-list)
                            ""
                            (function concat))))
  ;; jh added new function to process the technology
  ;;    option.  This function finds a technology in the
  ;;    option-list and sets the L5 technology.  If there
  ;;    is no technology in the option-list, the default
  ;;    nmos technology is used.
  (process-technology-option)
  ;; jh added new function to process the minimum feature
  ;;    size option.  This function finds a minimum feature
  ;;    size in the option-list and sets the L5 minimum
  ;;    feature size.  If there is no minimum feature size
  ;;    in the option-list, a default minimum feature size
  ;;    is used.
  (process-minimum-feature-size)
  ;; jh added a new function to process the magic option.
  ;;    This function sets or clears the magic flag and
  ;;    sets the scale factors for magic output.
  (process-magic-option)
  (cond ((member? 'int option-list)
         (cond ((null (catch (interpret file-name) note))
                (return ())))))
  (cond ((or (member? 'obj option-list)
             (member? 'cif option-list))
         (setq object (get-object file-name))
        (t (return t)))
  (cond ((null object) (return ())))
  (statistic (concat "Data-path has "
                     (length (object-data-path object))
                     " Units"))
```

```
    (cond ((member? 'obj option-list)
           (herald "Outputing .obj file")
           (setq obj
                 (make-object
                  (purge-library (object-definitions object)
                  (object-flags object)
                  (object-data-path object)
                  (object-control object)
                  (object-pins object)))
           (setq file (outfile (concat file-name ".obj")))
           (pp-form obj file)
           (close file)))
    (cond ((member? 'cif option-list)
           (setq item (catch (layout-object object) note))
           (cond ((null item) (return ())))
           (herald "Outputing .cif file")
           (cifout item file-name file-name)))
    (statistic (concat "Memory used - "
                       (/ (memory) 1024) "K"))
    (statistic (concat "Compilation took "
                       (quotient (- (car (ptime))
                                    (car initial-ptime))
                                 3600.0)
                       " CPU minutes"))
    (statistic (concat "Garbage collection took "
                       (quotient (- (cadr (ptime))
                                    (cadr initial-ptime))
                                 3600.0)
                       " CPU minutes"))
    (statistic (concat "For a total of "
                       (- \$gccount\$ initial-gccount)
                       " garbage collections"))
    (return t))))

(def process-command-line-options
 ;;  8 Apr 87 J Harmon added processing to process-option to
 ;;           remove options with a no prefix from the
 ;;           option-list if the option without the no
 ;;           prefix is specified on the command line.
 ;; 13 Apr 87 J Harmon moved all command line option
 ;;           processing to this new process-command-line-
 ;;           options function.
 (lambda (options)
  (do ((input-list options (cdr input-list)))
      ((atom input-list))
      ;; add option to option list and delete conflicting
      ;; options
      (add-delete-option (car input-list)))))

(def add-delete-option
 ;; 13 Apr 87 J Harmon renamed the old process-option to
;;          add-delete-option.  This function adds options
;;          to the option-list and deletes conflicting
;;          options.
(lambda (option)
 (cond ((not (atom? option))
        (warning "Option must be an atom"))
        ;Check for options that are not lists - if not,
        ;issue warning
       ((and (> (length (explode option)) 2)
             (equal 'n (car (explode option)))
             (equal 'o (cadr (explode option))))
             ;Check for options that start with "no"
        (delete-option (implode (cddr (explode option))))
        ;Remove option without "no" from option list
        (add-option option))
        ;; jh added processing to drop options that start
        ;;    with no from the option list if the option
        ;;    without the no is specified in the command
        ;;    line.
       ((member option (allowed-technologies))
       ;; Check if the option is a technology - if it is,
       ;; remove all other technologies and add this
       ;; technology.
        (do ((tech-list
             (allowed-technologies)
             (cdr tech-list)))
            ((atom tech-list))
            (delete-option (car tech-list)))
        (add-option option))
       ((and (equal 'u (car (last (explode option))))
       ;;Check if the option is a minimum feature size - if
       ;;it is, remove all other minimum feature sizes and
       ;;add this minimum feature size.
             (greaterp (length (exploden option)) 1)
             (greaterp (nth
                         (difference
                          (length (exploden option)) 2)
                         (exploden option))
                        47)
             (greaterp 58
                        (nth (difference
                              (length (exploden option))
                              2)
                             (exploden option))))
        (delete-minimum-feature-sizes)
        (add-option option)
       (t (add-option option)
          ;;Add all other options and delete any "no"
          ;;version.
          (delete-option (implode (append
                                             '(n o)
                                              (explode option))))))))

(def delete-option
 ;; 13 Apr 87 J Harmon added delete-option function that
 ;;           removes all instances of option from the
 ;;           option-list.
 (lambda (option)
  (cond ((member option option-list)
        ;Check if option is on the option-list - if it is,
        ;remove it.
         (setq option-list
               (nthdrop
                (iota option option-list)
                option-list))
         (delete-option option)))))

(def add-option
 ;; 13 Apr 87 J Harmon added add-option function that adds
 ;;           an option to the option list after first
 ;;           removing all instances of the option from the
 ;;           list.
 (lambda (option)
  (delete-option option)
  (setq option-list (cons option option-list))))

(def delete-minimum-feature-sizes
 ;; 13 Apr 87 J Harmon added delete-minimum-feature-sizes
 ;;           function that removes all minimum feature size
 ;;           specifications from the option list.
 (lambda ()
  (prog ()
   (do ((size-list option-list (cdr size-list))
        (size-option (car option-list) (car size-list)))
       ((atom size-list))
       ;finish when the size list is just ()
       (cond ((and (equal
                    'u
                    (car (last (explode size-option))))
                    ;Check if the option is a minimum
                    ;feature size - if it is, remove all
                    ;instances of it from the option list.
                   (greaterp
                    (length (exploden size-option))
                    1)
                   (greaterp (nth (difference
                                   (length
                                    (exploden size-option))
                                   2)
                                  (exploden size-option))
                        47)
                    (greaterp 58
                              (nth (difference
                                     (length
                                      (exploden size-option))
                                     2)
                                    (exploden size-option))))
               (delete-option size-option)
               ;remove the minimum feature size
               (delete-minimum-feature-sizes)
               ;use recursion after a minimum feature size
               ;has been removed and disable repetition if a
               ;minimum feature size has been removed.
               (return nil)))))))

(def process-technology-option
 ;; 14 Apr 87 J Harmon added new function process-
 ;;            technology-option that finds a technology in
 ;;            the option-list and sets the L5 technology.
 ;;            If there is no technology in the option-list,
 ;;            the default nmos technology is used.
 (lambda ()
  (do ((tech-list option-list (cdr tech-list))
       (tech 'nmos)) ;tech is reset by setq in the do loop
       ;;look for a technology in the option list - if a
       ;;technology is in the list, use it - if no
       ;;technology in the list, use nmos as default.
      ((atom tech-list) (technology! tech)
                        (statistic
                         (concat
                          "Using "
                          tech
                          " technology")))
      (cond ((member (car tech-list) (allowed-technologies))
             (setq tech (car tech-list)))))))

(def process-minimum-feature-size
 ;; 14 Apr 87 J Harmon added new function process-minimum-
 ;;            feature-size that sets a lambda spacing based
 ;;            on a minimum feature size specified on the
 ;;            command line or a default minimum feature size
 ;;            for the current technology.  This function
 ;;            verifies that the selected minimum feature
 ;;            size is valid for the current technology.
 (lambda ()
  (let ((min-size
         (do ((size-list option-list (cdr size-list))
              (size-option
               (car option-list)
               (car size-list))
              (size-found
               (default-minimum-feature-size)
```

```
                    ;;Check if the option is a minimum feature
                    ;;size - if it is set size-found to the new
                    ;;size
                    (cond ((and
                            (equal
                             'u
                             (car (last (explode size-option))))
                            (greaterp (length
                                       (exploden size-option))
                                      1)
                            (greaterp
                             (nth
                              (difference
                               (length (exploden size-option))
                               2)
                              (exploden size-option))
                             47)
                            (greaterp
                             58
                             (nth
                              (difference
                               (length (exploden size-option))
                               2)
                              (exploden size-option))))
                           size-option)
                          (t size-found))))
                  ((atom size-list) size-found))))
              ;; jh verify that the minimum feature size is valid
              (cond ((member min-size
                             (allowed-minimum-feature-sizes))
                     ;; jh set lambda spacing according to the
                     ;;    minimum feature size.
                     (lambda-spacing!
                      (convert-feature-size min-size))
                     (statistic
                      (concat "Minimum Feature Size " min-size)))
                    (t (note
                        (concat
                         "Invalid Minimum Feature Size "
                         min-size
                         " for technology "
                         (technology)
                         "."))))))))

(def process-magic-option
 ;; 14 Apr 87 J Harmon added new function process-magic-
 ;;           option that sets or clears the magic flag and
 ;;           sets the scale factors for CIF output.  The
 ;;           default is magic format.
 (lambda ()
  (cond ((member? 'nomagic option-list)
          (magic! ())
          (set-scale-factors 1 1))
         (t (magic! t)
            (cond ((eq 'nmos (technology))
                   (set-scale-factors 100 2)
                  (t (set-scale-factors 1 2)))))))
```

The following are changes to frame.l:

```
;;;;;>>> Add the following to the revision history for
;;;;;>>> frame.l:

;;; 16 Apr 87 J harmon modified the pad-class function to
;;;           make pad20b the default pad file for all
;;;           minimum feature sizes except 250 centimicrons.
;;;           The checking for valid lambda spacing was
;;;           moved to macpitts-compiler in prepass.l.
;;;           Replaced minimum-feature-size with lambda-
;;;           spacing in layout-object, conductivity-to-
;;;           power-bus-width, metal-thickness, and pad-
;;;           class functions to avoid confusion.  The
;;;           actual value used is the lambda spacing and
;;;           not the minimum feature size that is twice the
;;;           lambda spacing.

;;;;;>>> Replace the following functions:

(def layout-object
 ;; 16 Apr 87 J Harmon replaced minimum-feature-size with
 ;;           lambda-spacing in layout-object to avoid
 ;;           confusion.  The actual value used is the
 ;;           lambda spacing and not the minimum feature
 ;;           size that is twice the lambda spacing.
 (lambda (object)
  (prog (definitions flags data-path control pins gates
         straps conductivity power data-path-length
         control-length flags-length top-width bottom-width
         data-path-layout control-layout flags-layout
         river-layout wing-layout skeleton-layout
         internal-layout pins-layout ring-layout layout nets
         ring-width top-part bottom-part top-bank
         bottom-bank river-width bottom-part-river-points
         intended-right intended-top extended-right
         extended-top)
        (setq definitions (object-definitions object))
        (setq flags (object-flags object))
        (setq data-path (object-data-path object))
        (setq control (object-control object))
        (setq pins (object-pins object))
        (herald "Extruding gates")
        (setq gates (extrude-gates control flags))
```

```
(statistic (concat
             "Control has "
             (length gates)
             " columns"))
(cond ((member? 'opt-c option-list)
       (setq gates
        (nthelem-list
         (order (extrude-basic-straps gates)
                gates
                (count (length gates))
                (function junction-gate-number)
                (lambda (basic-strap)
                 basic-strap)
                (lambda (gate1 gate2)
                 (gate-before? gate1 gate2 gates))
                (lambda (gate1 gate2)
                 (gate-after? gate1 gate2 gates)))
        gates))))
(setq gates (insert-nor-ground-lines gates))
(herald "Extruding straps")
(setq straps (extrude-straps gates))
(statistic
 (concat
  "Circuit has "
  (slash-alpha
   (list
    (flags-transistor-count flags)
    (data-path-transistor-count
     data-path
     definitions)
    (control-transistor-count gates straps)
    (pins-transistor-count pins))
   0
   (function +)
   (lambda (x)
    (+ (car x) (cadr x))))
  " transistors"))
(statistic
 (concat
  "Control has "
  (slash-alpha straps
               0
               (function max)
               (function strap-track-number))
  " tracks"))
(setq
 conductivity
 (plus
  (data-path-conductivity data-path definitions)
  (control-conductivity gates straps)
  (flags-conductivity flags)))
```

```
(setq
 power
 (conductivity-to-power-bus-width conductivity 11))
(statistic
 (concat
  "Power consumption is "
  (conductivity-to-power-consumption
   (plus conductivity (pins-conductivity pins)))
  " Watts"))
(setq data-path-length
      (max
       (data-path-required-length
        data-path
        definitions)
       4))
(setq control-length
      (control-required-length gates))
(setq flags-length
      (max
       (flags-required-length flags power)
       4))
(setq top-width
      (max
       (data-path-required-width
        data-path
        power
        definitions)
       (flags-required-width flags power)))
(setq bottom-width
      (control-required-width straps))
(herald "Laying out data-path")
(setq data-path-layout
      (layout-data-path
       data-path
       power
       top-width
       definitions))
(herald "Laying out control")
(setq control-layout
      (layout-control
       gates
       straps
       power
       bottom-width))
(herald "Laying out flags")
(setq flags-layout
      (layout-flags
       flags
       power
       top-width))
(herald "Laying out river")
```

```
(setq top-part
      (merge
       (move data-path-layout (+ power 3) 0)
       (move
        flags-layout
        (+ power 3 data-path-length 3 power 3) 0)))
(setq bottom-part
      (move control-layout (+ power 3) (- power 4)))
(setq bottom-part-river-points
      (find-attributes bottom-part '(river)))
(setq top-bank
      (sort
       (alpha
        (lambda (point)
         (point-x (find top-part
                        (point-name point))))
        bottom-part-river-points)
       (function <)))
(setq bottom-bank
      (sort
       (alpha (function point-x)
              bottom-part-river-points)
       (function <)))
(setq river-width
      (+ (river-span 'NP 2 top-bank bottom-bank)
         (wing-span bottom-part)
         (- 4 power)))
(setq intended-top
      (+ power
         bottom-width
         power
         river-width
         (driver-width)
         power
         top-width
         power
         3
         power))
(setq intended-right
      (+ power
         3
         (max control-length
              (+ data-path-length
                 3
                 power
                 3
                 flags-length))
         3
         power))
(setq river-layout
      (river 'NP
             2
             (wing-span bottom-part)
             top-bank
             bottom-bank))
(herald "Laying out wing")
(setq wing-layout
      (layout-wing
       (sort
        (find-attributes bottom-part '(wing))
        (lambda (point1 point2)
         (< (point-x point1)
            (point-x point2))))))
(herald "Laying out skeleton")
(setq skeleton-layout
      (layout-skeleton
       power
       intended-top
       intended-right
       data-path-length
       bottom-width
       river-width))
(setq internal-layout
      (merge
       (move top-part
             0
             (+
               power
               bottom-width
               power
               river-width
               (driver-width)
               power))
       bottom-part
       (move (rotcw river-layout)
             0
             (+
              power
              bottom-width
              power
              river-width))
       (move wing-layout
             0
             (+ power bottom-width 4))
       skeleton-layout))
(herald "Laying out pins")
(setq pins-layout
      (layout-pins
       pins
       power
       intended-right
       intended-top
```

```
          (make-ring-width 0 0 0 0)
          (lookup-logo definitions)))
   (setq extended-right
         (extend-right pins intended-right))
   (setq extended-top
         (extend-top pins intended-top))
   (setq ring-width
         (get-ring-width
          (merge internal-layout pins-layout)
          extended-right
          extended-top))
   (setq pins-layout
         (layout-pins
          pins
          power
          intended-right
          intended-top
          ring-width
          (lookup-logo definitions)))
   (setq nets
         (append
          (extract-nets
           (merge internal-layout pins-layout)
           'top
           extended-right
           extended-top)
          (extract-nets
           (merge internal-layout pins-layout)
           'right
           extended-right
           extended-top)
          (extract-nets
           (merge internal-layout pins-layout)
           'bottom
           extended-right
           extended-top)
          (extract-nets
           (merge internal-layout pins-layout)
           'left
           extended-right
           extended-top)))
   (setq ring-layout
         (layout-nets nets
                      extended-right
                      extended-top
                      power
                      (find pins-layout '(power))
                      (find pins-layout '(ground))))
   (setq layout
         (first-quadrant
          (merge internal-layout
                          pins-layout
                          ring-layout)))
           (statistic (concat "Dimensions are "
                               ;jh replaced minimum-feature-size
                               ;   with lambda spacing
                               (quotient
                                (times (right layout)
                                       (lambda spacing))
                                100000.0)
                               " mm by "
                               (quotient
                                (times (top layout)
                                       (lambda spacing))
                                100000.0)
                               " mm"))
           (return layout))))

(def conductivity-to-power-bus-width
 (lambda (conductivity minimum-width)
  (max minimum-width
       (fix
        (plus 0.5
              (quotient
               (conductivity-to-power-consumption
                conductivity)
               (times (supply-voltage)
                      (maximum-metal-current-density)
                      ; jh replaced minimum-feature-size
                      ;    with lambda-spacing
                      (lambda-spacing)
                      (metal-thickness))))))))

(def metal-thickness                    ;in centimicrons
 ;; 16 Apr 87 J Harmon replaced minimum-feature-size with
 ;;           lambda-spacing to avoid confusion
 (lambda ()
  (times 0.4 (lambda-spacing))))

(def pad-class
 ;; 16 Apr 87 J Harmon made pad20b the default pad file for
 ;;           all minimum feature sizes except 250
 ;;           centimicrons.
 ;; 16 Apr 87 J Harmon replaced minimum-feature-size with
 ;;           lambda-spacing to avoid confusion
 (lambda ()
  (cond ((= lambda-spacing 250) 'rinout)
        (t 'pad20b))))
```

Make the following changes to Makefile:

Add the following to the list of modifications at the start of Makefile:

```
# Updated 22 Apr, 1987 by J. Harmon  Added magic to the
#               default option list in the macpitts.o section
```

Replace the (setq option-list line in the macpitts section with the following lines:

```
   (setq option-list
        '(magic opt-d opt-c stat obj cif herald nologo))
```

The following changes are to RVLSI-3 An Introduction to MacPitts:

On page 4 add the following options to the bottom of the list of possible <options>:

magic	nomagic*
nmos	scmos
cmos	hybrid
3u	2u
1.2u	

On page 5 add the following to the end of the paragraph at the top of the page:

> If the magic option is set, then the CIF output will be in a form that is compatible with the Magic program. If one of the nmos, scmos, cmos, or hybrid options is selected, the CIF output will be restricted to valid layers for that technology. The additional process width options of 3u, 2u, and 1.2u have been added for the scmos and hybrid technologies. If more than one technology or process width is specified, the last entry will be used. The process width must be a valid width for the technology specified. The default technology is nmos. The default minimum feature size for scmos and hybrid technologies is 3 microns and the default minimum feature size for nmos and cmos technologies is 4 microns.

RVLSI-5 L5 User's Guide by K.W. Crouch 7 March 1984 is not consistent with the version of L5 being used by MacPitts/MSC. The L5 User's Guide will have to be updated to conform to the version of L5 that is in use. The changes required by this scp should be developed at that time.

LIST OF REFERENCES

Arnold, K.C.R.C, Screen Updating and Cursor Movement Optimization: A Library Package, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA, not dated.

Brooks, R.A., Programming in Common Lisp, John Wiley & Sons, 1985.

Crouch, K.W., MIT Lincoln Laboratory additions to Franz Lisp, MIT Lincoln Laboratory, February 2, 1984.

Einspruch, N.G., VLSI Electronics Microstructure Science, v. 14, pp. 115-138, Academic Press, Inc., 1986.

Fairley, R.E., Software Engineering Concepts, pp. 32-33, McGraw-Hill, 1985.

Feldman, S.I., Make - A Program for Maintaining Computer Programs, Bell Laboratories, Murray Hill, NJ, August 15, 1978.

Glasser, L.A. and Dobberpuhl, D.W., The Design and Analysis of VLSI Circuits, p. 12, Addison-Wesley Publishing Company, 1985.

Hon, R.W and Sequin, C.H., A Guide to LSI Implementation, 2d ed., XEROX Palo Alto Research Center, SSL-79-7 January 1980.

Kelley-Bootle, S., The Devil's DP Dictionary, McGraw-Hill Book Company, 1981.

Langdon, G.G.Jr., Computer Design, Computeach Press Inc., p. 522, 1982.

Malagon-Fajar, M.A., Silicon Compilation Using a Lisp-Based Layout Language, M. S. Thesis, Naval Postgraduate School, Monterey, CA, June 1986.

McGilton, H. and Morgan, R., Introducing the UNIX System, pp. 33-83, McGraw-Hill Book Company, 1983.

MIT Lincoln Laboratory Project Report RVLSI-3, An Introduction to MacPitts, by Southard, J.R., 10 February 1983.

MIT Lincoln Laboratory Project Report RVLSI-5, L5 User's Guide, by Crouch, K.W, 7 March 1984.

Shelly, D., "Made to Order", Digital Review, March 1986, pp. 128-131.

Suppes, P., Introduction to LOGIC, p. 34, D. Van Nostrand Company, Inc., 1957.

University of California, Berkeley, CA, Computer Science Division (EECS) Report No. UCB/CSD 86/272, 1986 VLSI Tools: Still More Works by the Original Artists, by Scott, W.S., and others, December 1985.

UNIX Programmer's Manual, 4.2 Berkeley Software Distribution, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA, August 1983.

Weinberger, A., "Large Scale Integration of MOS Complex Logic: A Layout Method", IEEE Journal of Solid-State Circuits, v. SC-2, No. 4, pp. 182-190, 4 December 1967.

Weste, N.H.E. and Eshraghian, K., Principles of CMOS VLSI Design, pp. 160-189 and 310-378, Addison-Wesley Publishing Company, October 1975.

BIBLIOGRAPHY

Carlson, D.J., Application of a Silicon Compiler to VLSI Design of Digital Pipelined Multipliers, MSEE Thesis, Naval Postgraduate School, Monterey, CA, June 1984

Larrabee, R.C., VLSI Design with the MacPitts Silicon Compiler, MSEE Thesis, Naval Postgraduate School, Monterey, CA, September 1985.

Mullarky, A.J., CMOS Cell Library for a Silicon Compiler, MSEE Thesis, Naval Postgraduate School, Monterey, CA, March 1987.

INDEX

1+ 46, 83
acceptance criteria 59
adder 153, 154
alpha 168
always 29, 32, 44, 50
analysis phase 6, 59
AND 28, 30, 31, 168
APL 39
architecture 3, 8, 32, 60, 62
arg line 51, 52
argument 11
argument list 167
argv 166
arithmetic 11
assign-bus-numbers 50
back-quote 165
baseline 141, 168
benchmark 161
bit 10, 11, 16, 46, 50, 53-55, 81, 83, 89, 93, 148-152
bit line 51, 52
bit slice 10, 16
bounding box 143
branch 22-25
buffer 94, 96, 100, 118, 134
bus 8
C 38, 39
c-routines.c 38, 39, 57, 165
c-routines.o 39, 57, 165
call 25-27, 46, 47, 50, 84, 92
capability 59
capacitance 79
carry input line 150
cascade voltage switch logic 62, 71, 72, 77-79, 90
cell 7, 60, 139, 141-144, 147, 148, 159
cfasl 165
chip 167
CIF 3, 33, 38, 40, 52, 56, 140-143, 161, 166
cif option 41
cifout 38, 40, 52
cifsymbol 143, 144
circuit description 1
clock 66-68, 71-73
clock cycle 17, 24
clock cycle update 167
clock edge 20
clocked 92
clocked circuit 17
clocked CMOS 62, 68, 90
clocked memory element 17
clocked? 167
CMOS 62, 64, 65, 68-75, 77-80, 89, 90
column 17, 51, 52
combinational logic 17-21, 25, 27, 80, 81, 83, 86-88, 93
command line option 36, 41, 44, 51, 52, 140
complementary logic 62, 64, 65, 68, 69, 71, 74, 75, 77-80, 89, 90
component 44, 45, 50
component-list 50
computer program 1
cond 29, 88, 89, 95, 96, 114-116, 132, 145, 147, 148, 154-158, 161
CONDbuffer 118, 120, 133-135
CONDfeed 115, 116, 132, 134
conditional 94-96, 99, 100, 114, 116, 120, 132
CONDnopass 115, 118, 132, 134
CONDpass 115, 116, 118, 132, 134
CONDtap 115-118, 132, 134
CONDtrue 115, 132, 134, 147, 148
connection 42
constant 31, 41, 42, 50, 54, 149
constant substitution 36, 41
contact 60

control 38, 145, 158
control line 51, 52, 81, 82
control logic array 8, 11, 16, 17, 43, 44, 50-53, 55, 82, 93, 97, 145, 158
control signal 16, 87-89
control-data-path 86
control.l 53, 55, 144, 147, 148, 158, 166-168
control.o 57
controller 1, 5-8, 28, 32, 58-63, 72, 73, 77, 80, 83, 86, 89, 138, 152, 154, 161
controller data path 86, 87
controller organization 6
controller.l 142
counter-no-stack 45, 46, 49, 81, 83, 85, 161
counter-stack 45-47, 84, 85, 161
CPU 44
cross reference file 57
CRYSTAL 140-142, 161
cr_tty.o 165
current state 25, 27
curses 38, 39, 57, 165
cursor movement 39
cutoff 60
data 55
data path 8, 10, 11, 16, 32, 38, 43, 45, 50-54, 61, 80, 81, 83, 86, 87, 89, 93-95, 97, 138
data path internal bus 43, 51, 53, 54
Data Path Unit 91
data path word size 25, 27, 28, 82, 83, 86, 87, 167
data primitive 33, 50
data processing 8, 10, 27
data signal 16
data storage 10, 11
data structure 10, 27, 28, 34-36, 39-45, 50-53, 58, 61
data-path 86, 145, 147-151, 154
data-path.l 42, 43, 45-47, 53, 54, 86, 98, 166, 167
data-path.o 57
debug 52, 139, 141
debugger 168
DecodeEven0 103, 104, 109, 111, 122, 125-127, 130
DecodeEven1 103-105, 109, 111, 122, 125-127, 130
DecodeLSB 103, 104, 121, 125
DecodeOdd0 103, 108, 109, 111, 122, 125, 130
DecodeOdd1 103, 108, 109, 111, 122, 125, 129, 130
definition 35, 41, 44, 47, 145, 167
defstruct 35, 36, 39, 40, 98
defstructs.l 36, 40, 44, 45, 50, 86, 166, 167
defstructs.o 40, 57
defsymbol 45-47, 53-55, 86, 142, 143
defun 167
DeMorgan's Laws 31
depletion 30
design 59
design decision 59, 80
design phase 6, 59, 62
design rule checker 141, 142
design specification 3, 25, 26, 31, 33, 35, 36, 39, 41, 42, 44, 46, 47, 50, 56, 57, 60, 81, 88, 89, 92, 94, 96, 114, 132, 138, 161
destination definition 47, 50
determine-number-of-tracks 168
development process 1, 6
dimension 42
display function 43, 44
domino logic 62, 69-73, 90
drain 79
drive current 96

drive line 51, 52
dual 64, 65, 71, 73, 78
dumplisp 38
dynamic CMOS 62, 66-69, 71. 80, 81, 90
electrical characteristic 42
enhancement 29, 30
env 167
.env file 167
environment 141, 167
err 165
ESIM 31, 140-142, 161, 162
eval 35, 36, 41-43
evaluation 66-68, 70, 71
EvenBuffer 118, 122, 135, 137
EvenConnect 104, 111, 125, 131
EvenInverter 103, 108, 111, 125, 130
exclusive or 74
expand-form 42
ext2sim 141, 142, 161
external node 34
external signal 148
extract definition 41
extract-component 50
extract-component-list 44
extract.l 38, 41, 44, 51, 86, 144, 145, 166, 167
extract.o 57
extraction 36, 40, 41, 44, 50, 51, 58, 61
extraction file 141, 142
fabrication 140
fasl 40
file name extension 5
file type 5
FILO 92
FILO stack 93
find-number-of-tracks 168
finite state machine 1, 8, 17, 92
fixnum 165
flag 8, 10, 16, 27, 28, 33, 34, 38, 88, 145, 167
flags.l 53, 55, 166
flags.o 57
floor plan 8, 86, 92, 96, 142, 159
frame.l 38, 41, 51, 52, 53, 55, 166, 167
frame.o 57
Franz Lisp 38, 39, 139, 141, 143, 165, 166, 167
front-page.l 40, 166, 167
FSM 1, 17-19, 21-23, 25, 27, 28, 32, 61, 68, 80, 81, 86-88, 138
function 42, 50
function definition 41, 42, 56
functional 8
functional simulation 31, 32, 36, 39, 41, 42, 56, 57
functional simulator 141
garbage collection 44
gate 17, 51
general.l 36, 43, 54, 56, 166, 167, 168
general.o 57
get-library 42
get-object 38
get-object1 44
get-sequencers-from-component-list 44, 45, 47
get-sequencers-required-definitions 44, 47
global variables 40
GND 8, 29, 31
go 27, 46, 47, 49, 50, 81-83, 92
goal 59, 60, 62, 63, 77-80
grade-predicate 165
graph 22
ground 8, 17
growth 60
herald 44
hold time 20, 21
home 143
homogeneous structure 63
hybrid 140
hybrid structure 63
if 50
implementation phase 7, 59
include 40
increment 83, 93

incrementer 25, 27, 32, 46, 60, 93, 152
installation 139
instance 1
instantiate 42, 54
int option 41
integer 25
integration 7
interpreter 167
internal 149
internal bus 11, 149-154
internal node 34
internal port 10, 11, 16, 27, 28, 46, 50, 53, 54, 81-84, 93, 149, 151, 156-158
internal port definition 47, 49, 50
internal signal 16, 83, 85, 148
internal signal definition 47, 49
internal structure 58, 60
interpret 41, 56
interpret.l 36, 39, 41, 42, 56, 167
interpret.o 57
inverter 29-31, 43, 69, 73, 76, 130-132, 134-137
ISI 57, 58, 139, 165, 166, 168
item 36, 38, 40, 52, 53, 55
L5 52, 53, 166
L5-err 166
L5.l 36, 38, 40, 52, 53
L5.o 40, 57
--L5-symbol-list 144
label 143
lambda 143
latch up 60
layout 1, 2, 6-8, 10, 25, 45, 54-56, 60, 140-144, 147, 149
layout aid 43
layout-control 53, 55
layout-data-path 45-47, 53, 98
layout-flags 53, 55
layout-gen-form 54
layout-label 40
layout-mpx 54
layout-mpx0 54
layout-mpx1 54
layout-mpx2 54
layout-mpx3 54
layout-object 38, 41, 51-53, 55
layout-organelle 43, 54
layout-pad 55
layout-pins 53
layout-port-output 54
layout-rom0 142
layout-rom1 142
layout-singleton-operand-list 54
layout-skeleton 53
layout-unit 53, 54
layout-weinberger-gates 55
layout-weinberger-straps 55
layout-wing 53
layout-data-path 86
libcurses.a 165
library 5, 35, 36, 41-43, 50, 56, 57, 60, 158, 163, 168
library definition 41
libtermcap 165
LIFO 49, 85, 86
lincoln.l 38, 39, 40, 57, 165, 168
lincoln.o 40
Lisp 3, 5, 35, 38, 40, 42, 57, 143, 144
list 35, 39, 50, 165
Liszt 39, 139, 165
Logfile 168
logic 11, 28, 31
logic delay 89
logic structure 6, 62, 63
logical function 42
lookup-gen-form 54
LSBbuffer 118, 121, 135, 136
macpit 57, 58
MacPitts 3, 33, 42, 57, 139, 140, 165, 168
macpitts-compiler 36, 41, 52, 56
macpitts-directory 42

macro 35, 50, 143, 144, 158, 165, 168
macro definition 41, 42
macro expansion 36, 41
Magic 40, 43, 139-142, 161
main 56
maintenance phase 7
make 57
make clean 55, 57
make doc 58
make macpitts 55, 57
make pads.l 55
make pads.o 55
make xref 57
Makefile 38, 55, 57, 168
mapcar 168
mark 53
Mealy FSM 18, 19, 21, 24, 25, 32, 60, 63-65, 68, 70, 72, 73, 76, 77, 90, 138
memory 34
microprocessor 5
microprogram ROM 62, 80, 87-89, 91-94, 96, 97, 100, 102, 103, 117, 120-123, 125, 138, 141, 145, 149, 154, 156, 158
microprogrammed controller 4, 5, 8, 40, 139, 144, 145, 159, 161
minimum feature size 56, 143
modified domino logic 62, 72, 73, 91
Moore FSM 18-22, 24, 68, 70-73
MOS 60
MSC 3, 6, 8, 10, 17, 25-29, 31-36, 38-41, 52, 57-61, 80, 81, 86-88, 92, 93, 97, 138, 139, 141-144, 148, 158, 159, 161-164
multiplexer 11, 16, 25, 27-32, 50, 51, 53, 54, 61, 82, 84, 87, 149-155
n-type transistor 64
NAND 43, 125, 127, 129, 132, 133, 168
new-states 167
next address logic 4
next state 92, 151
next state number 27, 82, 151
nMOS 3, 6, 8, 17, 28, 29, 31, 32, 43, 56, 58, 61, 64, 80, 81, 93, 97, 140, 145, 158, 166
no-counter-no-stack 45, 46, 47, 49, 81, 83, 152, 153, 161
node 22, 23, 33, 34
noherald 44
NOR 17, 30, 43, 50, 125, 126, 132, 133, 159
nostat 44
NOT 29, 30, 31, 148
note 44
number-of-tracks 168
number-tracks 168
object 144, 145, 148, 158, 159
object 36, 38, 40-42, 44, 52, 58, 61
object file 57
.obj file 36, 38, 42
.obj 86
OddBuffer 118, 122, 135, 137
OddConnect 104, 111, 125, 131
OddInverter 103, 108, 111, 125, 130
operand 11, 44
opt 52
opt-c 51
opt-d 51
opt-n 52
opt-p 52
opt-s 52
optimization 51, 52, 63
Opus 165, 166, 167
OR 30
order 51, 52
order-this 52
order.l 51, 167, 168
order.o 57
organelle 10, 11, 16, 42, 43, 46, 50, 53, 54, 80, 81, 83-85, 87, 93, 147-150, 152, 153

organelle definition 41-43
organelles-track-number 168
organelles.l 43, 163, 167, 168
organelles.o 36, 42, 43, 57, 60, 168
organization 62, 80, 89, 91
ospeed 165
output port 10, 11, 16, 50, 53, 54, 149, 153
p-type transistor 64
package 8
pad 8, 11, 16, 34, 38, 53, 55, 56, 94, 95, 153
pad20b 56
padgen.l 55, 56, 166
padgen.o 57
pads.l 55-57, 166
pads.o 57, 166
par 28, 50
performance 59
permutation 52
physical 8
pin 145
PLA 17, 62, 80, 81
place 88
polysilicon 60
pop 25
port 29, 33, 34, 50, 87, 167
port definition 50
port-internal 148
port-output 148, 153
port-state-driver 167
port-state-value 167
post-process 44, 50, 51
power 8, 60
pp-form 165
precharge 66, 67, 70, 71, 73, 76
predicate 165
prepass.l 36, 38, 41, 42, 44, 52, 56, 86, 166
prepass.o 57
process 27, 29, 31, 44-47, 50, 81, 83, 84, 87, 89, 93, 94, 99, 104, 138, 145-147, 150-152, 155-157, 161
programmable logic array 17, 62, 80
propagate update 167
$prpr 165
pseudo-nMOS 62, 64-66, 77-81, 90
pull down 29, 30, 32, 64-66, 70, 71, 77-79
pull up 30, 32, 60, 64, 65, 71, 73, 76-79
push 25
reset 150, 151, 153
read-cif 56
reader 168
recursion 26, 27
register 10, 16, 19-21, 27-29, 31, 33, 34, 45, 47, 50, 53-55, 81-87, 93, 148-158
register definition 47, 49, 50
regular 63
remove 165
remove-parameter 165
requirement 7, 8, 32, 33, 58-63, 65, 68, 70, 72, 73, 77, 80, 159
reset 82
return 25-27, 46, 47, 50, 84
return state 25
rinout 56
river 54
river router 43
ROM 4, 62, 80, 88, 89, 145, 154-158
ROM array 94-96, 99, 100, 102, 104, 112, 114, 115, 117, 118, 120, 123, 132
ROM0 100, 103, 118, 123, 142
ROM1 100, 102, 118, 123, 142
ROMnull 100, 103, 123, 125
ROMtest.mag 141, 142
route 88
row decoder 89, 94, 96, 100, 103, 104, 108, 117, 118, 125, 146, 147

RowBuffer 104, 114, 118, 132, 135
run 60
saturation 60
SCMOS 3, 6, 7, 31, 32, 40, 43, 59, 60, 62, 100, 126, 127, 130, 131, 134-137, 139, 140, 166
segment 52
sequencer
sequencer 45-47, 49, 50, 81-87, 93, 97, 138, 148, 150-153, 159, 161
setq 50
setup time 19, 21
side effect 31
signal 8, 16, 17, 21, 23, 27-30, 33, 34, 47, 49, 82, 87, 93-96, 99, 152, 154, 156, 161, 167
signal definition 47, 50
signal-state-driver 167
signal-state-value 167
silicon compiler 1, 2, 3, 138, 140, 141, 165
simulation file 141, 142, 161
simulation 140
simulator 140-142, 167
single bit data-path 144, 148, 158
single bit processing 10, 16
single bit storage 10
source 79
source definition 47, 50
special 165
stable 19, 20, 21, 167
stack 25, 32, 60, 85, 86
stack depth 26, 85
stack overflow 26
standard cell 1, 62, 80, 87, 88, 91
stat 44
state 17, 19, 21-28, 34, 46, 47, 49, 50, 88, 89, 94, 145, 146, 155-157, 167
state number 25, 27, 32, 45, 60, 82, 83, 86, 89, 92-94, 99, 103-105, 108, 111, 113, 122, 131, 138, 150
state number memory 82, 85, 86
state transition 19-25, 34, 46, 68, 81-83, 92, 150
state transition diagram 21, 22, 24
state variable 17-19, 21, 23, 25, 32, 60
static circuit check 141
static electrical check 142
statistic 44
stepper 168
structural requirement 63
structure 62-66, 70, 71, 73, 74, 77, 78, 80, 88-90
subroutine 26, 27
substate 60, 94, 88, 147, 148, 155, 157, 158
superbuffer 43, 135, 136
switch level simulation 31
symbol 36, 40, 53, 55, 86, 144
taxi 145, 146, 148, 149, 168
technology 59, 140, 141, 143, 144, 163
technology dependent 43, 60, 92, 163, 164
technology independent 7, 60, 92, 100
termcap 165
terminal display 39
test 7, 42, 50, 139, 141, 142, 159, 161
test definition 41, 42, 56
test input 147, 148
test line 51, 52, 95, 147, 150, 152, 153, 154
test phase 7, 59
test signal 94, 100, 115, 138, 149
thing 165
this-node-column 52
this-segment-nodes 52
timing 140, 141

topCONDfeed 115, 116, 132, 134
tr-traceexit 165
track 17, 51
track-number 168
transistor 29, 30, 60
transmission gate 29, 62, 67, 68, 73-80, 91
Two-State 104, 111, 112, 125, 131, 147
unit 10, 11, 16, 32, 45-47, 50-54, 60, 81-87, 149-159
unit number 50
UNIX 38, 39, 42, 57, 139
unset 167
/usr/macpit 165, 168
validation 7, 42, 59
Vdd 8, 30, 31
verification 7
vi 38
VLSI 1, 2, 8, 33, 34, 59, 60, 139-141
vsplicing-macro 168
warning 44
Weinberger array 17, 62, 80, 145, 158
well 60
wired AND 29, 31
wired logic 32
word 83
word-length 167
word-size 33
workstation 57, 58
XOR 17, 74, 168
xref 57, 168

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Department Chairman, Code 62 Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5000	2
4.	Dr. D.E. Kirk, Code 62KI Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5000	5
5.	Dr. H.H. Loomis, Code 62LM Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5000	3
6.	Dr. S. Michael, Code 62MI Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5000	1
7.	Prof. R. Cristi, Code 62CR Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5000	1
8.	Mr. P. Blankenship Massachusetts Institute of Technology Lincoln Laboratory P.O. Box 73 Lexington, Massachusetts 02173-0073	1
9.	Mr. Mr. J. O'Leary Massachusetts Institute of Technology Lincoln Laboratory P.O. Box 73 Lexington, Massachusetts 02173-0073	1

9.	Mr. Mr. J. O'Leary Massachusetts Institute of Technology Lincoln Laboratory P.O. Box 73 Lexington, Massachusetts 02173-0073	1
10.	Commander, Naval Ocean Systems Center Code 552 (Mr. Charles Morrin) 200 Catalina Blvd. San Diego, California 92152-5000	1
11.	Dr. T. Bestul Naval Research Laboratory, Code 7590 4555 Overlook Ave SW Washington, D.C. 20375	1
12.	Dr. A. Ross Naval Research Laboratory, Code 9110 4555 Overlook Ave SW Washington, D.C 20375	1
13.	CDR David Southworth Office of Naval Technology, Code ONT227 800 N. Quincy (BT #1) Arlington, Virginia 22217-5000	1
14.	Mr. James Hall Office of Naval Technology, Code ONT20P4 800 N. Quincy (BT #1) Arlington, Virginia 22217-5000	1
15.	Dr. D. O'Brien Lawrence Livermore National Laboratory P.O. Box 5504, L-156 Livermore, California 94550	1
16.	Mr. A. DeGroot Lawrence Livermore National Laboratory P.O. Box 808, L-156 Livermore, California 94550	1
17.	CAPT E. Malagon 1200 Forsome Lane Virginia Beach, Virginia 23462	1
18.	Commanding Officer Naval Electronic Systems Engineering Center Vallejo (Code 530E) Vallejo, California 94592-5017	1